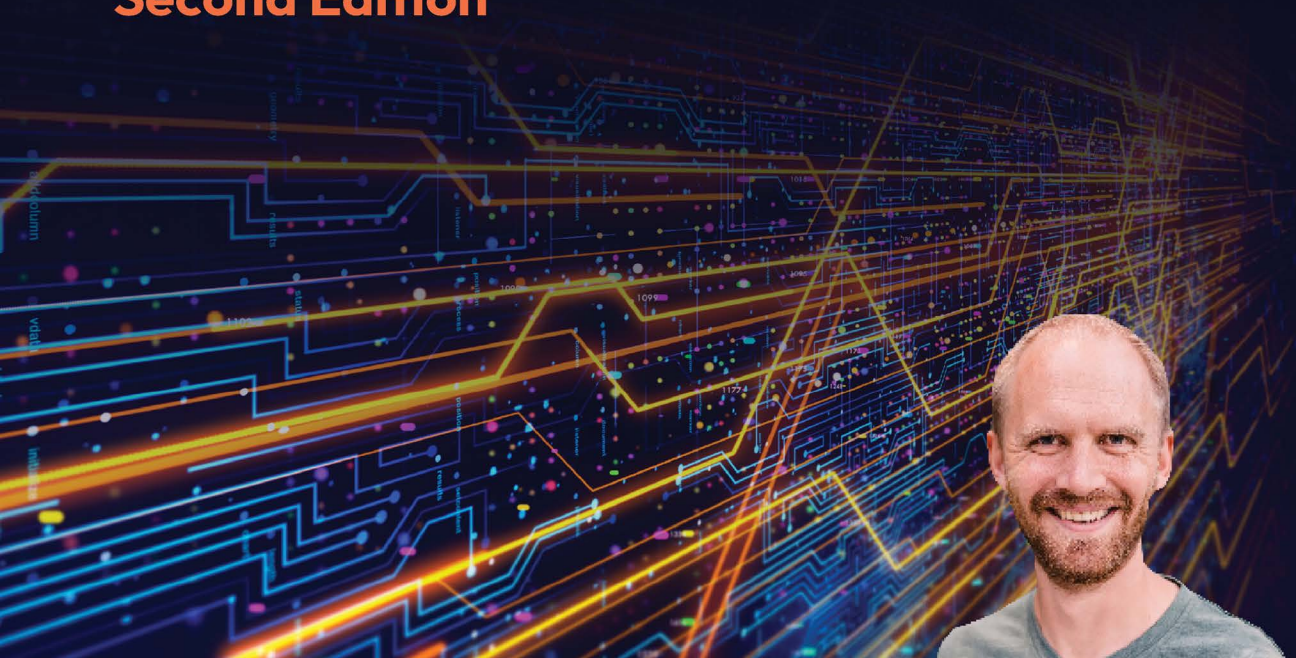


EXPERT INSIGHT

# API Testing and Development with Postman

API creation, testing, debugging, and  
management made easy

**Second Edition**



**Dave Westerveld**

**<packt>**

# API Testing and Development with Postman

Second Edition

API creation, testing, debugging, and management  
made easy

**Dave Westerveld**



# API Testing and Development with Postman

## Second Edition

Copyright © 2024 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Senior Publishing Product Manager:** Denim Pinto

**Acquisition Editor – Peer Reviews:** Gaurav Gavas

**Project Editor:** Meenakshi Vijay

**Senior Development Editor:** Elliot Dallow

**Copy Editor:** Safis Editing

**Technical Editor:** Anjitha Murali

**Proofreader:** Safis Editing

**Indexer:** Pratik Shirodkar

**Presentation Designer:** Rajesh Shirsath

**Developer Relations Marketing Executive:** Vipanshu Parashar

First published: April 2020

Second edition: June 2024

Production reference: 1130624

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN: 978-1-80461-790-8

[www.packt.com](http://www.packt.com)

# Contributors

## About the author

**Dave Westerveld** is passionate about sharing his knowledge and expertise to help testers stay relevant in the ever-changing world of software. He has helped many software testers through his several popular video courses and has shared his expertise at conferences, online talks, and podcasts. He has also worked in the software industry for many years and in many different roles. He has worked, among other roles, as an exploratory tester, a test automation developer, and an API integrations developer.

*I would like to thank my wife, Charlene, for her unwavering support of me in everything that I do. You are the rock that makes it possible for me to show up in the world in the ways that I do. You are always excited for me to share my expertise with the world. Thank you for your love and support.*

## About the reviewers

**Christina Thalayasingam** is a software engineering people manager at Northwestern Mutual with 9 years of industry experience. She holds a bachelor's degree in computer science engineering and has worked for companies such as Sysco and Dassault Systèmes®. She has a passion for testing, and beyond her daily work, she has been an active speaker at conferences and meetups such as Star East, Women in Tech Global Conference, and TestCon Europe.

**Neil McCormick** is a quality assurance manager who has worked in the QA space since 1998. He began API testing in the early 2000s. He helped pioneer the testing methodologies of his company, AAA, and in 2020 was promoted to his current position. He believes testers should never stop learning, exploring, and most importantly growing – both professionally and personally.

*I am grateful to the author, Dave Westerveld, and the Packt Publishing team, my bosses Jerry and Kristi, my team, and too many people in the realm of family and friends to call out.*

*To my wife, Nandi, thank you for the weekends and nights you sacrificed for me to be able to partake in this opportunity. You smiling at my boyish excitement when I really had fun reviewing a chapter and listening to me speak endless technobabble to you has been greatly appreciated!*

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.com/invite/nEN6EBYPq9>





# Table of Contents

<b>Preface</b>	<b>xix</b>
<hr/>	
<b>Chapter 1: API Terminology and Types</b>	<b>1</b>
<hr/>	
What is an API? .....	2
Types of API calls .....	3
Installing Postman .....	5
Starting Postman • 6	
Setting up a request in Postman • 6	
Saving a request • 7	
The structure of an API request .....	7
API endpoints • 8	
API actions • 9	
API parameters • 10	
Request parameters • 10	
Query parameters • 10	
API headers • 11	
API body • 13	
API response • 13	
Learning by doing – making API calls .....	14
Setting up the test application • 15	
Making a call to the test application • 16	
A challenge • 17	



<b>Considerations for API testing .....</b>	<b>17</b>
Beginning with exploration • 17	
<i>Exploratory testing case study</i> • 18	
Looking for business problems • 20	
Trying weird things • 21	
<b>Different types of APIs .....</b>	<b>21</b>
REST APIs • 21	
SOAP APIs • 22	
<i>SOAP API example</i> • 23	
GraphQL APIs • 26	
GraphQL API example • 26	
<b>Summary .....</b>	<b>28</b>
 <b>Chapter 2: API Documentation and Design .....</b>	 <b>31</b>
<b>Technical requirements .....</b>	<b>32</b>
<b>Start with the purpose .....</b>	<b>32</b>
Figuring out the purpose of an API • 33	
<i>Personas</i> • 33	
<i>The why</i> • 33	
<i>Try it out</i> • 34	
<b>Creating usable APIs .....</b>	<b>35</b>
Usable API structure • 35	
Good error messages • 36	
<b>Documenting your API .....</b>	<b>36</b>
Documenting with Postman • 37	
Good practices for API documentation • 40	
RESTful API Modeling Language • 42	
<b>API design example .....</b>	<b>42</b>
Case study – Designing an e-commerce API • 42	
<i>Defining the endpoints</i> • 43	

---

<i>Defining the actions</i> • 44	
<i>Adding query parameters</i> • 45	
<i>Using the RAML specification in Postman</i> • 46	
Modeling an existing API design • 47	
<b>Summary</b> .....	<b>48</b>
 <b>Chapter 3: OpenAPI and API Specifications</b>	 <b>49</b>
<hr/>	
Technical requirements .....	50
What are API specifications? .....	50
API specification terminology • 51	
Defining API schema • 51	
Types of API specifications • 52	
<i>RAML</i> • 52	
<i>API Blueprint</i> • 52	
<i>OpenAPI/Swagger (OAS)</i> • 53	
Creating an OAS .....	53
Parts of an OAS • 54	
Petstore OAS schemas • 58	
Creating your own OAS .....	58
Starting the file • 59	
<i>Understanding the API schema</i> • 61	
Defining parameters • 63	
Describing request bodies • 64	
Using examples • 64	
Using API specifications in Postman .....	65
Creating a mock server • 66	
Validating requests • 68	
<b>Summary</b> .....	<b>68</b>
 <b>Chapter 4: Considerations for Good API Test Automation</b>	 <b>71</b>
<hr/>	
Technical requirements .....	72

<b>Exploratory and automated testing .....</b>	<b>72</b>
Exercise – considerations for good API test automation • 74	
Writing good automation • 74	
Types of API tests • 75	
<b>Organizing and structuring tests .....</b>	<b>77</b>
Creating the test structure • 77	
Organizing the tests • 79	
<i>Environments • 80</i>	
<i>Collection variables • 81</i>	
<i>Choosing a variable scope • 82</i>	
<i>Exercise – using variables • 87</i>	
<b>Creating maintainable tests .....</b>	<b>87</b>
Using logging • 87	
Test reports • 88	
<b>Creating repeatable tests .....</b>	<b>89</b>
<b>Summary .....</b>	<b>90</b>
 <b>Chapter 5: Understanding Authorization Options .....</b>	 <b>93</b>
<b>Understanding API security .....</b>	<b>94</b>
Authorization in APIs • 95	
Authentication in APIs • 95	
<b>API security in Postman .....</b>	<b>96</b>
Getting started with authorization in Postman • 97	
Using Basic Auth • 98	
Using bearer tokens • 100	
Using API keys • 101	
Using AWS Signature • 102	
Using OAuth • 103	
<i>Setting up OAuth 2.0 in Postman • 104</i>	
<i>OAuth 1.0 • 108</i>	

- Digest authentication • 109
- Hawk authentication • 111
- Using NTLM authentication • 112
- Using Akamai EdgeGrid • 113
- Handling credentials in Postman safely • 114
- Summary ..... 114
- Chapter 6: Creating Test Validation Scripts 117**
- Technical requirements ..... 118
- Checking API responses ..... 118
  - Checking the status code in a response • 119
    - Using the *pm.test* method • 120
    - Using Chai assertions in Postman • 121
    - Try it out • 122
  - Checking the body of a response • 122
    - Checking whether the response contains a given string • 122
    - Checking JSON properties in the response • 123
    - Try it out • 126
  - Checking headers • 127
  - Custom assertion objects in Postman • 127
  - Creating your own tests • 130
    - Try it out • 130
  - Creating folder and collection tests • 130
  - Cleaning up after tests • 131
- Setting up pre-request scripts ..... 132
  - Using variables in pre-request scripts • 133
  - Passing data between tests • 133
  - Building request workflows • 135
    - Looping over the current request • 136
    - Running requests in the collection runner • 138

---

<b>Using environments in Postman .....</b>	<b>140</b>
Managing environment variables • 140	
<b>Summary .....</b>	<b>142</b>
 <b>Chapter 7: Data-Driven Testing .....</b>	 <b>145</b>
 <b>Technical requirements .....</b>	 <b>146</b>
<b>Defining data-driven testing .....</b>	<b>146</b>
Setting up data-driven inputs • 148	
Thinking about the outputs for data-driven tests • 148	
<b>Creating a data-driven test in Postman .....</b>	<b>150</b>
Creating the data input • 150	
Adding a test • 152	
Comparing responses to data from a file • 154	
<b>Challenge – data-driven testing with multiple APIs .....</b>	<b>157</b>
Challenge setup • 157	
Challenge hints • 157	
<b>Summary .....</b>	<b>158</b>
 <b>Chapter 8: Workflow Testing .....</b>	 <b>161</b>
 <b>Different types of workflow tests .....</b>	 <b>161</b>
Linear workflows • 162	
Business workflow • 164	
<b>Workflow testing with the Flows feature in Postman .....</b>	<b>165</b>
Configuring a Send Request block • 166	
Building a Flow in Postman • 167	
<b>Advice for creating workflow tests .....</b>	<b>171</b>
Checking complex things • 171	
Checking things outside of Postman • 172	
<b>Summary .....</b>	<b>173</b>

---

<b>Chapter 9: Running API Tests in CI with Newman</b>	<b>175</b>
<b>Technical requirements</b> .....	<b>176</b>
<b>Getting Newman set up</b> .....	<b>176</b>
Installing Newman • 176	
<i>Installing Node.js</i> • 177	
<i>Using npm to install Newman</i> • 178	
Running Newman • 178	
<b>Understanding Newman run options</b> .....	<b>181</b>
Using environments in Newman • 181	
Running data-driven tests in Newman • 183	
Other Newman options • 184	
<b>Reporting on tests in Newman</b> .....	<b>185</b>
Using Newman's built-in reporters • 186	
Using external reporters • 187	
<i>Generating reports with htmlextra</i> • 188	
Creating your own reporter • 188	
<b>Integrating newman into CI/CD builds</b> .....	<b>192</b>
General principles for using Newman in CI/CD builds • 192	
Example – using GitHub Actions • 193	
<b>Summary</b> .....	<b>198</b>
 <b>Chapter 10: Monitoring APIs with Postman</b>	 <b>199</b>
<b>Setting up a monitor in Postman</b> .....	<b>200</b>
Creating a monitor • 200	
Using additional monitor settings • 203	
<i>Receive email notifications for run failures and errors</i> • 203	
<i>Retry if run fails</i> • 204	
<i>Set request timeout</i> • 204	
<i>Set delay between requests</i> • 205	

---

<i>Follow redirects • 205</i>	
<i>Enable SSL validation • 206</i>	
Adding tests to a monitor • 206	
<b>Viewing monitor results .....</b>	<b>208</b>
Cleaning up the monitors • 211	
<b>Summary .....</b>	<b>212</b>
 <b>Chapter 11: Testing an Existing API .....</b>	 <b>213</b>
<b>Finding bugs in an API .....</b>	<b>213</b>
Setting up an API for testing • 214	
Testing the API • 215	
Finding bugs in the API • 217	
Resetting the service • 218	
Example bug • 219	
<b>Automating API tests .....</b>	<b>219</b>
Reviewing API automation ideas • 220	
Setting up a collection in Postman • 220	
Creating the tests • 221	
<b>An example of automated API tests .....</b>	<b>222</b>
Setting up a collection in Postman • 222	
Creating the tests • 226	
<i>Updating the environment • 226</i>	
<i>Adding tests to the first request • 228</i>	
<i>Adding tests to the second request • 229</i>	
<i>Adding tests to the POST request • 232</i>	
<i>Cleaning up tests • 233</i>	
<i>Adding tests to the PUT request • 235</i>	
<i>Adding tests to the DELETE request • 236</i>	
<b>Sharing your work .....</b>	<b>237</b>
Sharing a collection in Postman • 238	
<b>Summary .....</b>	<b>239</b>

---

**Chapter 12: Creating and Using Mock Servers in Postman** **241**

---

**Getting started with mock servers** ..... 241

What is a mock server? • 241

When to use a mock server • 242

Things to be careful of with mock servers • 243

**Setting up mock servers in Postman** ..... 244

Modifying mock server values • 245

Creating more mock values • 246

Mocking route parameters • 247

Mocking dynamic data • 248

**Using mock servers** ..... 251

Using private servers • 251

Mocking a third-party API • 252

**Summary** ..... 254

---

**Chapter 13: Using Contract Testing to Verify an API** **255**

---

**Understanding contract testing** ..... 256

What is contract testing? • 256

How to use contract testing • 257

Who creates the contracts? • 258

*Consumer-driven contracts* • 258*Provider-driven contracts* • 259**Setting up contract tests in Postman** ..... 260

Creating a contract testing collection • 261

Adding tests to a contract test collection • 264

*Running contract tests* • 267*Using Postman Interceptor* • 269**Running and fixing contract tests** ..... 271

Fixing contract test failures • 271

Sharing contract tests • 272

**Summary** ..... 273



---

## **Chapter 14: API Security Testing** **275**

---

<b>OWASP API Security list .....</b>	<b>275</b>
Authorization and authentication • 275	
Broken object-level authorization • 277	
Broken property-level authorization • 279	
Unrestricted resource consumption • 280	
Unrestricted access to business workflows • 281	
Unsafe consumption of APIs • 281	
<b>Fuzzing .....</b>	<b>282</b>
Fuzz testing with Postman • 283	
Cleaning up the tests • 287	
Fuzzing with built-in methods in Postman • 290	
<b>Summary .....</b>	<b>291</b>

## **Chapter 15: Performance Testing an API** **293**

---

<b>Different types of performance load .....</b>	<b>294</b>
Processing load • 294	
Memory load • 295	
Connection load • 296	
<b>Using load profiles in Postman .....</b>	<b>297</b>
Fixed load profile • 297	
Spike load profile • 299	
Ramp load profile • 301	
Endurance load profile • 303	
<b>Running performance tests in postman .....</b>	<b>304</b>
Running multiple requests • 306	
<b>Performance testing considerations .....</b>	<b>309</b>
When to do performance testing • 309	
Benchmarking • 310	

---

Repeatability • 311	
Collaboration and communication • 312	
Summary .....	313
<b>Other Books You May Enjoy</b>	<b>319</b>

---

<b>Index</b>	<b>323</b>
--------------	------------

---



# Preface

In a world of fast food, fast fashion, and fast-to-market strategies, does quality even matter? The pressures of the world we live in seem to push us towards taking shortcuts, even when it comes to producing high-quality software. I am doing my part to push back against that world. I think quality matters. We have enough easily broken junk in our lives. It's time for more quality.

This book is one small stake that I've put in the ground in an attempt to help the world see more high-quality software. I hope that whether you are a professional tester, or a developer looking to learn more about testing, you will be able to join me in my attempt to improve the world through quality software applications.

APIs are becoming the backbone of the internet. They help companies to communicate with each other externally and also provide the communication infrastructure for many internal pieces of modern software systems. A marriage is held together by good communication, and so it is with the internet too. Good communication between different services is important to well functioning applications. For that reason, API testing matters for producing good-quality software.

On the surface, this book is primarily about the API testing tool Postman, but I have also tried to weave in examples and teachings that will help you to use that tool in a way that will have a real impact on quality. If you work through this book, you will gain an in-depth grasp of how Postman works, and you will also have a solid foundation in how to think about API testing in general. I want you to have more than just the ability to manipulate Postman to do what you want it to. I also want you to be able to know when and how to use it so that you can be an effective part of creating high-quality APIs.

## Who this book is for

The first person I write for is myself. A lot of the ideas I talk about in this book are things I was learning myself a few years ago. In fact, Postman comes out with new capabilities so often that some of what is in this second edition are new things I learned while writing the book.

I'm always growing and learning and I love sharing what I've learned with others to help them along on their journey.

Getting started with API testing can be overwhelming. It's a huge topic and it can be intimidating to get started, so I wrote this book primarily for those software testers and developers who find themselves needing to test an API and not knowing where to start. Throughout this book I try not to assume much in-depth programming experience although an understanding of some of the basics of programming will certainly help with some sections of the book.

If you are working as a software tester and you are interested in expanding your skills into API testing, this book is certainly for you. If you are a developer who is looking to enhance your skills around testing and quality, congratulations, you are setting yourself up for a successful career! Developers that know and understand how to produce good quality software will always be in high demand. Whatever your background, you may be able to skim through some parts of this book, but if you spend some time with it, you will find that you come away knowing how to use Postman and how to design and write good API tests.

## What this book covers

*Chapter 1, API Terminology and Types*, gets you started with some of the basic API terminology and introduces you to the different types of APIs.

*Chapter 2, API Documentation and Design*, covers the design principles that apply to creating and testing APIs, and both *how* and *why* to create useful documentation.

*Chapter 3, Open API and API Specifications*, explains what API specifications are and how to get started with using them in Postman.

*Chapter 4, Considerations for Good API Test Automation*, teaches you how to create and execute valuable and long-lasting API tests in Postman.

*Chapter 5, Understanding Authorization Options*, walks through how to use many of the API authorization methods available in Postman.

*Chapter 6, Creating Test Validation Scripts*, explains how to create and use test scripts in Postman.

*Chapter 7, Data-Driven Testing*, discusses what data-driven testing is and how to use it to create scalable tests in Postman.

*Chapter 8, Workflow Testing*, explains what workflow tests are and how to create flows in Postman.

*Chapter 9, Running API Tests in CI with Newman*, shows how to run Postman API tests at the command line with the Newman runner.

*Chapter 10, Monitoring APIs with Postman*, explains how to monitor product usage of APIs with Postman monitoring.

*Chapter 11, Testing an Existing API*, works through a hands-on example that shows what kind of tests to create when testing an existing API.

*Chapter 12, Creating and Using Mock Servers in Postman*, explains what mock servers are and how to set up and use them in Postman.

*Chapter 13, Using Contract Testing to Verify an API*, discusses what contract testing is and shows how to create and use contract testing in Postman.

*Chapter 14, API Security Testing*, gives a brief introduction to security testing and gives an example of setting up fuzz testing in Postman.

*Chapter 15, Performance Testing an API*, explains the different types of performance testing and walks through some of the features in Postman that can be used to assess API performance.

## To get the most out of this book

This book is intended to equip you with skills that you can use immediately in your work as a tester or developer. If you want to get the most value that you can out of this book, put the things that you learn into practice as soon as you possibly can. Work through all the exercises in this book, but also try to take the ideas that you learn and put them into practice in the “real world” as well.

This book does not assume a lot of prior knowledge of APIs, or even development and testing principles. As long as you have a basic grasp of web technology and what software development looks like in general, you should be able to follow along with this book and pick up everything that you need. Some of the test scripts in Postman use Javascript, but you don’t need to know much about how that works in order to follow along, although a basic understanding would be helpful. There are examples and challenges throughout the book. They are an important part of the book and in order to get the most out of it, you should take the time to work through them.

## Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/gbp/9781804617908>.

## Conventions used

There are a number of text conventions used throughout this book.

**CodeInText:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “The /product endpoint gives information about the products accessed by this API.”

A block of code is set as follows:

```
openapi: 3.0.1
info:
  title: ToDo List API
  description: Manages ToDo list Tasks
  version: "1.0"
servers:
  -url: https://localhost:5000/todolist/api
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
/carts:
  post:
  get:
    queryParameter:
    username:
  /{cartId}:
    get:
    put:
```

Any command-line input or output is written as follows:

```
npm install -g newman
```

**Bold:** Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: “Click on the **Import** button and choose the **OpenAPI** option.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

## Get in touch

Feedback from our readers is always welcome.

**General feedback:** Email [feedback@packtpub.com](mailto:feedback@packtpub.com) and mention the book’s title in the subject of your message. If you have questions about any aspect of this book, please email us at [questions@packtpub.com](mailto:questions@packtpub.com).

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packtpub.com/submit-errata>, click **Submit Errata**, and fill in the form.

**Piracy:** If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.



## Share your thoughts

Once you've read *API Testing and Development with Postman, Second Edition*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781804617908>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.



# 1

## API Terminology and Types

Learning something new can feel a little like falling over the side of a ship. Everything is moving and you can barely keep your head above water. You are just starting to feel like you understand how something works and then a new piece of knowledge comes out of nowhere, and your whole world feels topsy-turvy again. Having something solid to hold on to gives you the chance to look around and figure out where you are going. This can make all the difference in the world when learning something new.

In this chapter, I want to give you that solid foundation. As with almost any specialty, API testing and development has its own terminology. There are many terms that have specialized meanings when you are working with APIs. I will be using some of those terms throughout this book, and I want to make sure that you and I share a common understanding of what they mean.

As much as possible, I will use standard definitions. Some terms, however, do not have clearly agreed-on definitions, and for those terms, I'll share how I intend to use and talk about them in this book. Be aware that as you read or listen to things on the internet (or even just interact with teammates), you may come across others who use the terms in slightly different ways.

This book is not a dictionary, so I don't intend to just write down a list of terms and their definitions. That would be boring and probably not all that instructive. Instead, I'll spend a bit of time on the theory of what an API is and how you test it. I will weave in explanations and definitions of important terminology throughout the text.

This chapter will cover the following main topics:

- What is an API?
- Types of API calls

- Installing Postman
- The structure of an API request
- Considerations for API testing
- Different types of APIs

By the end of this chapter, you will be able to use Postman to make API requests and have a good grasp of basic API terminology. You will also have the opportunity to work through an exercise that will help you cement what you are learning, allowing you to start to use these skills in your day-to-day work.

## What is an API?

A 1969 NASA publication entitled *Computer Program Abstracts* contains a summary of a real-time display control program sold by IBM (only \$310! Plus \$36 if you want the documentation). The advertisement says that this program was designed as an operator-application programming interface – in other words, an API.

**Application Programming Interfaces (APIs)** have been around for about as long as computer code has. Conceptually, it is just a way for two different pieces of code (or a human and some code) to interface with each other. A class that provides certain public methods that other code can call has an API. A script that accepts certain kinds of input has an API. A driver on your computer that requires programs to call it in a certain way has an API.

However, as the internet grew, the term *API* narrowed in focus. Almost always now, when someone talks about an API, they are talking about a web API. That is the context I will use in this book. A web API takes the concept of an interface between two things and applies it to the client/server relationship that the internet is built on. In a web API, a client is on one side of the interface and sends requests, while a server (or servers) is on the other side of the interface and responds to the request.

Over time, the internet has changed and evolved, and web APIs have changed and evolved along with it. Many early web APIs were built for corporate use cases, with strict rules in place as to how the two sides of the interface could interact with each other. A type of API called the **Simple Object Access Protocol (SOAP)** was developed for this purpose. However, in the early 2000s, the web started to shift toward becoming a more consumer-based place. Some of the e-commerce sites, such as eBay and Amazon, started to publish APIs that were more public and flexible. This was followed by many social media sites, including Twitter, Facebook, and others. Many of these APIs were built using a pattern called **Representational State Transfer (REST)**.

This approach is more flexible than SOAP and is built directly on the underlying protocols of the internet.

The internet continued to change though, and as mobile applications and sites grew in popularity, so did the importance of APIs. Some companies faced challenges with the amount of data they wanted to transfer on mobile devices, so Facebook created yet another type of API called GraphQL. This type of API defines a query language that helps to reduce the amount of data that gets transferred, while also introducing a slightly more rigid structure to the API. Each of these different API types works well in some scenarios, and I will explain more about what they are later in the chapter. However, before I get into the details of each of these types of APIs, it is important to first understand some of the concepts that underpin all web API calls.

## Types of API calls

Some calls to APIs can change things on the server, while others return data without changing anything. The terms **safe** and **idempotent** are used to describe the different ways that API calls can affect data. These terms might sound a bit intimidating, so in order to better understand them, let's look at an illustration that uses something we can all understand: LEGO pieces.

Imagine that there is a table with a couple of LEGO pieces on it and I'm sitting by the table. I represent an API, while the table represents a server, and the LEGO pieces represent objects. If you come along and want to interact with the LEGO, you must do so through me. In this illustration, the LEGO pieces represent objects on a server, I represent an API, and you represent a client. In picture form, it looks something like this:

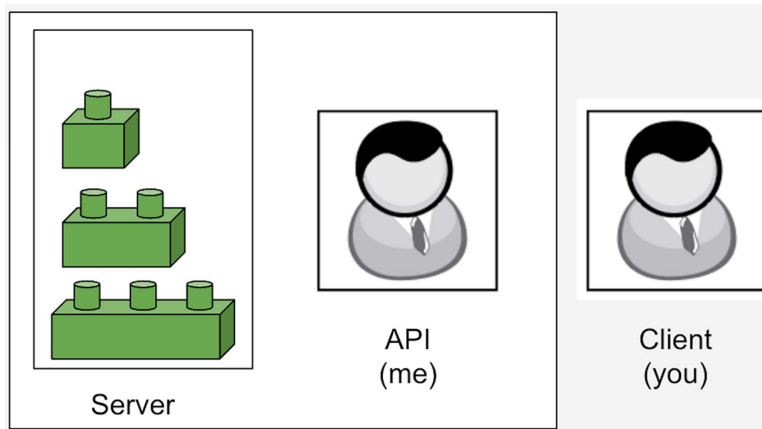


Figure 1.1: Representation of a server and a client connected by an API

You are going to be the client in this imaginary relationship. This means you can ask me to do things with the LEGO. You ask me to tell you what size the top LEGO piece is. I reply that it has a size of one. This is an example of an API request and response that is **safe**. A safe request is one that does not change anything on the server. By asking me for information about what is going on in the server, you have not changed anything on the server itself.

There are other kinds of API calls though. Imagine that you gave me a brick with a size of two and asked me to replace the top brick on the stack with the one you gave me. I do that, and in doing so I have changed the server state. The brick stack is now made up of a brick with a size of three, followed by two bricks with a size of two, as shown in the following diagram:

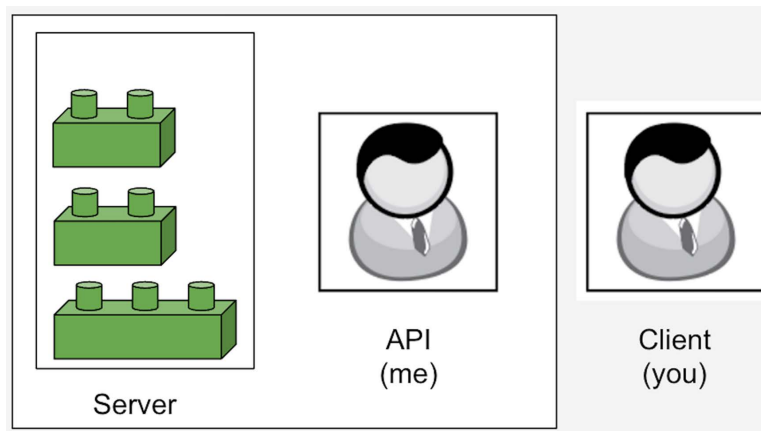


Figure 1.2: New server state

Since the server has changed, this is not a safe request. However, if you give me another brick with a size of two and ask me to once again replace the top brick with the brick you just gave me, nothing will change on the server. The stack will still be made up of a brick with a size of three followed by two bricks with a size of two. This is an example of an **idempotent** call. API calls that return the same result no matter how many times you call them are known as idempotent.

Let's imagine one more type of call. In this case, you give me a brick and ask me to add it to the top of the stack. I do that and now we have a stack of four bricks. This is clearly not a safe call, since the server has changed, but is it idempotent?

The answer is no, but take a second to think about it, and make sure you understand why this call would not be idempotent.

If you are struggling to see why it is not idempotent, think of what happens if you repeat the same request. You give me another brick and you ask me to add it to the top of the stack. If I do that a second time, is the brick stack still the same as it was after the first time you added it? No, of course not! It now has five bricks and every additional brick you give to me to add to the stack will change it. An idempotent call is one that only changes things the first time you execute it and does not make any changes on subsequent calls. Since this call changes something every time, it is not idempotent.

Safety and idempotency are important concepts to grasp, especially when it comes to testing APIs. For example, if you are testing calls that are safe, you can run tests in parallel without needing to worry about them interfering with each other. But if you are testing calls that are not safe or idempotent, you may need to be a little more careful about what kinds of tests you run and when you run them.

There are a few more important terms that we need to learn about, and we also need to dig into the structure of API requests. However, it will be a lot easier to do that if we have something concrete to look at, so at this point, let's take a brief pause to install Postman and send our first request.

## Installing Postman

Postman can be run on the web or as a desktop application. The functionality and design are similar between the web and desktop applications, but in this book, I will mostly use the desktop application, so I would recommend you install it too. The app is available for Windows, Mac, and Linux, and installing it is the same as pretty much any other program you've installed. However, I would highly recommend creating a Postman account if you don't already have one. Creating an account is totally free, and it makes it a lot easier to manage and share your work. The free account of Postman is very generous in what functionality it enables. Postman does have some enterprise or advanced-level features that require a paid account, but all the examples in this book will work with the free features of Postman. However, if you don't have an account at all, it will be difficult to follow along with some of the examples, so I would strongly recommend that you register for one:

1. Go to <https://postman.com>.
2. Choose the **Sign Up for Free** option.
3. Create an account, and when asked how you want to use Postman, choose the **Download Desktop App**, and then install it as you would any program on your computer.



If you already have a Postman account but don't yet have the desktop app, you can skip steps two and three and just download it directly from the Postman home page.

I will be using the Mac version of Postman, but other than the occasional screenshot looking a bit different, everything should be the same regardless of which platform you are running Postman on.

I will primarily use the desktop application in this book, but there are times when the web application is helpful. I would recommend that you set it up as well. During the sign-up process, you can download the Desktop Agent. This agent allows the web version of Postman to get around some of the cross-origin restrictions that web browsers put on web requests. These restrictions are very important for security on the web, but when testing, we often need to be able to work around them, and this agent will allow you to do that. Once again, you can download and install this as you would any other program.

## Starting Postman

Once you have Postman installed, open the application. The first time you open Postman, it will ask you to sign in. Once you've signed in, you will see the main screen with a bunch of different options for things that you can do with Postman. Don't worry about all these options right now. We will cover all of them (and more) as we go through this book. For now, just note that they are there and that you can do a lot of cool stuff with Postman. Maybe even get a little bit excited about how much you are going to learn as you go through this book!

## Setting up a request in Postman

It's time to set up an API call so that we can dissect it and see how it all works:

1. To set up a new request, you can click on the + near the top of the application to add a new tab.

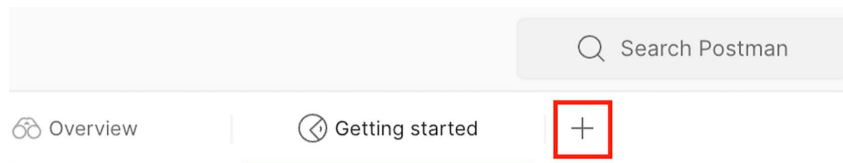


Figure 1.3: Create a new request tab

For this first example, I will use the Postman Echo API. This is an API Postman provides that can be used to do some simple testing.

2. In the field that says **Enter URL or paste text**, type in the following URL: `https://postman-echo.com/get`.
3. Click on the **Send** button to the right of the URL field, and you should see a response from the server. Don't worry about the actual data of the response; for now, just congratulate yourself for having sent your first request with Postman!

## Saving a request

Now that you have created a request, let's look at how to save requests in Postman. Postman requires that requests be saved into something called Collections. Collections are a way to collect multiple requests that belong together:

1. Click on the **Save** button above and to the right of the request URL.

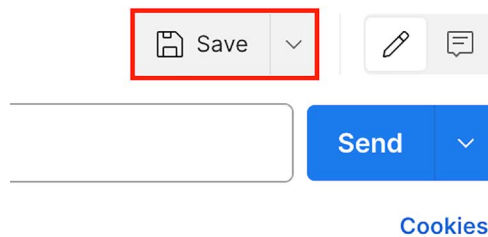


Figure 1.4: Save a request

2. In the pop-up dialog, give the request a name. Call it `Postman Echo GET`.
3. Click on the **New Collection** option at the bottom of the popup.
4. Name the collection something like `Postman Echo Requests` and click **Create**.
5. Click the **Save** button on the dialog. This will create the collection for you and save the request into that collection.

Now that you have saved a request, let's start digging into the basic structure of an API request.

## The structure of an API request

The request tab in Postman provides a lot of information about the various pieces that make up an API request. Each of these pieces plays an important part in sending and receiving data with an API, so I will walk you through each one in turn. Some parts of an API request are optional, depending on what kind of request it is and what you are trying to do with it, but there are three pieces that are required for every API request. Every API request needs an endpoint, headers, and an action; let's look at those next.

## API endpoints

Every web-based API request must specify an **endpoint**. In the **Postman requests** tab, you are prompted to enter the request URL. Postman asks you to enter a URL because an API endpoint is just a URL. We use the term *URL* so much that we can sometimes forget what it stands for. URL is an acronym for **Uniform Resource Locator**. The endpoint of an API call specifies the resource, or the “R” of the URL. In other words, an API endpoint is a uniform locator for a particular resource that you want to interact with on the server. URLs help you to locate resources on a server, so they are used as the endpoints in an API call.

In order to understand this, let’s set up a concrete example:

1. Create a second collection by clicking on the **New** button above the navigation panel and choosing **Collection** on the popup.
2. Name the collection something like **GitHub API Requests**.
3. If you expand the collection, you will see a prompt telling you that the collection is empty, and a link reading **Add a request** is provided. Click on that link.

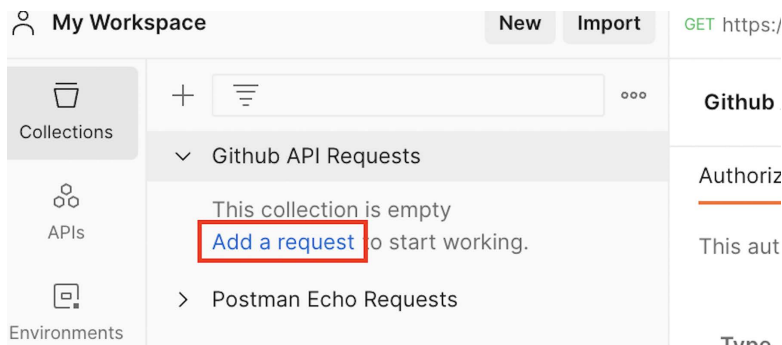


Figure 1.5: Add a request to a collection

4. Name the request **Get Repos** and fill in the request URL field with the following URL: `https://api.github.com/users/djwester/repos`.
5. Click **Send** to see the response.

This endpoint will give you information about my public repositories on GitHub. If you have a GitHub account of your own, you can enter your username in the part of the URL where it says `djwester` and get back data for your own repositories.

You will often see an API endpoint specified without the base part of this API. So, for example, if you look at the GitHub API documentation, it will report the endpoint for this as `/users/:username/repos`. All the GitHub API calls start with the same base URL (in other words, `https://api.github.com`), so this part of the endpoint is often left out when talking about an endpoint. If you see API endpoints listed that start with a `/` instead of with `http` or `www`, just remember that you need to go and find the base API URL for the endpoint in order to call it.

## API actions

Every API call needs to specify a resource that we work with. This resource is the endpoint, but there is a second thing that every API call needs. An API needs to do something with the specified resource. We specify what we want an API to do with API **actions**. These actions are sometimes called **verbs**, and they tell the API call what we expect it to do with the resource that we have given it. For some resources, only certain actions are valid, while for others, there can be multiple different valid API actions.

In Postman, you can select the desired action using the drop-down menu beside the textbox where you entered the URL. By default, Postman sets the action to **GET**, but if you click on the dropdown, you can see that there are many other actions available for API calls. Some of these actions are specialized for particular applications, so you won't run into them very often. In this book, I will only use **GET**, **POST**, **PUT**, and **DELETE**. Many APIs also use **PATCH**, **OPTIONS**, and **HEAD**, but using these is very similar to using the four that I will use, so you will be able to easily pick up on how to use them if you run into them. The rest of the actions in this list are not often used, and you will probably not encounter them much in the applications that you test and create.

The four actions (**GET**, **POST**, **PUT**, and **DELETE**) are sometimes summarized with the acronym **CRUD**. This stands for **Create**, **Read**, **Update**, and **Delete**. In an API, the **POST** action is used to create new objects, the **GET** action is used to read information about objects, the **PUT** action is used to modify (or update) existing objects, and (surprise, surprise) the **DELETE** action is used to delete objects. In practice, having an API that supports all aspects of CRUD gives you the flexibility to do almost anything you might need to, which is why these four actions are the most common ones you will see.

API actions and endpoints are required for all web APIs, but there are several other important pieces to API requests that we will consider.

## API parameters

API parameters are used to create structure and order in an API. They organize similar things together. For example, in the API call that we looked at earlier, we get the repositories for a particular user in GitHub. There are many users in GitHub, and we can use the same API endpoint to get the repository list for any of them by merely changing the username in the endpoint. The part of the endpoint that accepts different usernames is a **parameter**.

### Request parameters

The username parameter in the GitHub repositories API endpoint is known as a **request parameter**. You can think of a request parameter as a replacement string in the API endpoint. They are very common in web APIs. You will see them represented in different ways in the documentation of different APIs. For example, the GitHub documentation uses a colon in front of the request parameter to indicate that it is a request parameter and not just another part of the endpoint. You will see endpoints specified like this in the GitHub documentation:

```
/users/:username/repos.
```

In other APIs, you will see request parameters enclosed in curly braces instead. In that case, the endpoint would look like this:

```
/users/{{username}}/repos.
```

Whatever the format used, the point of request parameters is to get information about different objects that are all the same type. We have already seen how you can do that with this endpoint by replacing my username with your username (or any other GitHub user's name).

### Query parameters

There is another kind of parameter that you can have in an API endpoint. This kind of parameter is known as a **query parameter**, and it is a little bit trickier to deal with. A query parameter often acts like a kind of filter or additional action that you can apply to an endpoint. It is represented by a question mark in the API endpoint and is specified with a key, which is the item you are querying for, and a value, which is what you want the query to return.

That's all very abstract, so let's look at it with the GitHub request we sent earlier. This endpoint supports a couple of different query parameters. One of them is the type parameter. In order to add parameters to an API endpoint in Postman, make sure you have the **Params** tab selected, and then enter the name of the query parameter into the **Key** field and the value into the **Value** field. In this case, we will use the type parameter, so enter that word into the **Key** field.

For this endpoint, the type parameter allows us to filter based on whether you are the owner of a repository or just a member. By default, the endpoint will return only those repositories that I am the owner of, but if I want to see all the repositories that I am a member of, I can put member in the **Value** field for this. At this point, the request should look something like this:

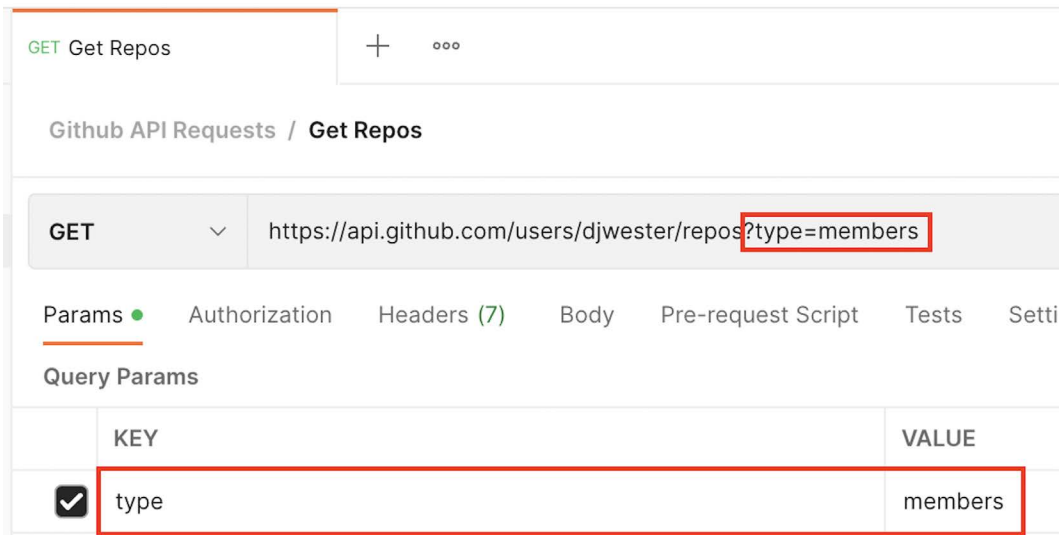


Figure 1.6: Query parameter type in an API call

You can see that when you specify query parameters in the **Params** tab, Postman automatically adds them to the request URL. If I send this request, I get back all the repositories that I am a member of, as opposed to just the ones that I own. Parameters are a powerful API paradigm, but there are still a few more fundamental pieces of the API structure that I haven't talked about yet. The next thing we will look at are API headers.

## API headers

Every API request needs to include some **headers**. Headers include some of the background information that is often not that important to human users, but they help the server have some information about the client that is sending the request. Sometimes, we will need to modify or add certain headers in order to get an API to do what we want, but often, we can just let the tool that we are using send the default headers that it needs to send without worrying about it.

In Postman, you can see what headers will be sent with your request by using the **Headers** tab. When you first go to that tab, it might look like there aren't any headers specified. In that case, you probably need to toggle the view to show the auto-generated headers:

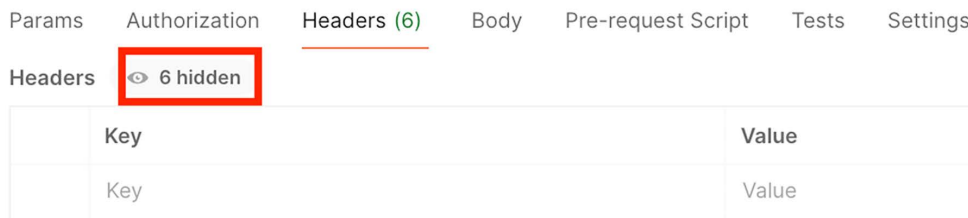


Figure 1.7: Headers hidden

Once the headers are revealed, you should see a list that looks something like this:

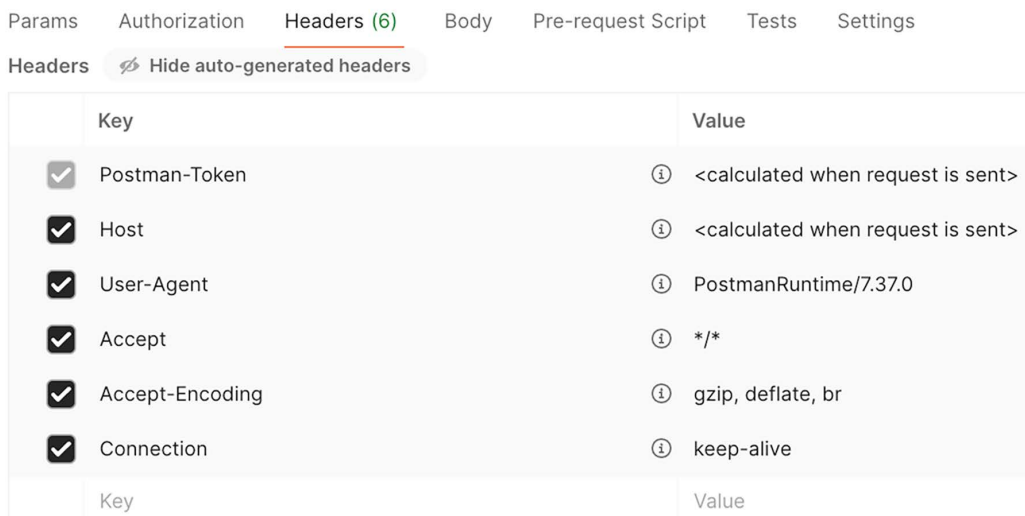


Figure 1.8: List of headers

You can modify the auto-generated headers and add additional ones here as needed. I will go into more detail on how headers work and how to use them in future chapters, so for now, you don't need to worry about them too much. The point of mentioning them here is just to make sure you know the terminology. Let's turn our attention instead to the body of an API request.

## API body

If you want to create or modify resources with an API, you will need to give the server some information about what kind of properties you want the resource to have. This kind of information is usually specified in the **body** of a request.

The request body can take on many forms. If you click on the **Body** tab in the Postman request, you can see some of the different kinds of data that you can send. You can send form-data, encoded form data, raw data, binary data, and even GraphQL data. As you can imagine, there are a lot of details that go into sending data in the body of a request. For example, if you were to try and make a new repository using the GitHub API, you might enter a body that looks something like this:

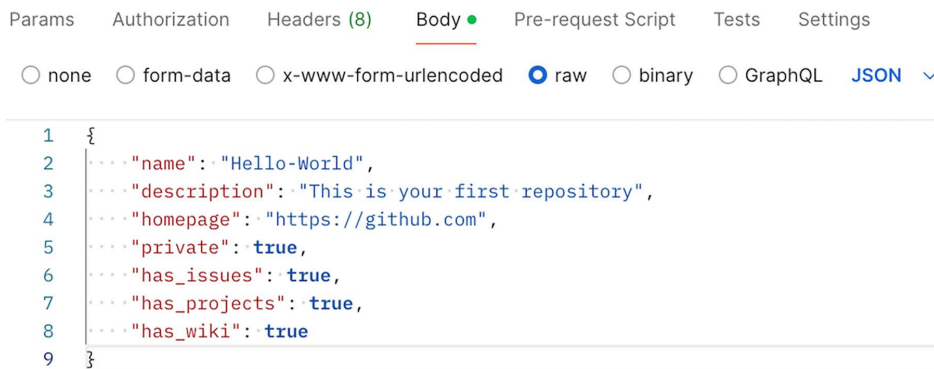


Figure 1.9: Body of a request

Most of the time, GET requests do not require you to specify a body. Other types of requests, such as POST and PUT, which do require you to specify a body, often require some form of authorization, since they allow you to modify data. We will learn more about authorization in *Chapter 5, Understanding Authorization Options*. Once you can authorize requests, there will be a lot more examples of the kinds of things you might want to specify in the body of an API request.

## API response

So far, we have spent a lot of time talking about the various pieces that make up an API request, but there is one very important thing that we have been kind of ignoring. An API is a two-way street. It sends data to the server in the request, but then the server processes that request and sends back a response.



The default view that Postman uses displays the response at the bottom of the request page. You can also modify the view to see the request and the response in side-by-side panels. You can change to this view if you so wish by clicking on the **Two pane view** icon at the bottom of the application, as shown in the following screenshot:

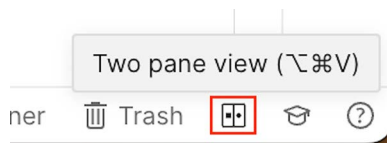


Figure 1.10: Switching views

There are a few different aspects to the response. The most obvious one is the body of the response. This is usually where most of the information that you are looking for will be included. In the GitHub repositories requests that you have made, the lists of repositories will show up in the body of the response, and Postman will display them in that tab.

An API response can also include a few other things, such as cookies and headers. These kinds of things can be very important hints as to what is going on when testing or creating APIs, and I will talk about them more as we go through the book.

We have covered a lot of ground when it comes to how API requests work. We have seen how an API request can consist of a lot of different pieces. These simple pieces all come together to create a powerful tool. You now have a grasp of the basic pieces that make up an API call and how to use them in Postman. It's almost time to talk about how to use this to test APIs, but before we get into that, I want to pause for a minute so that you can put into practice all this theory that I've just gone over.

## Learning by doing – making API calls

Books are a great way to grow and learn. You are reading this book, so I don't think I need to convince you of that! However, reading a book (or even three or four books) on a topic does not mean you understand that topic. There is theory and then there is putting it into practice. These are two very different things, and if all you do is read about a topic, you will feel like you know that topic, but that will be more of a feeling than a reality.

If you want that to be a reality and you don't just want this book to be another piece of theoretical knowledge bouncing around inside your head, you need to put into practice the things that you are learning.

Hands-on exercises might feel like they are slowing down your reading, but working through them will actually help you learn faster and get more out of this book. But enough talking about it. Let's get to some exercises!

In order to help you with getting some hands-on experience, I have created a simple API application that you can interact with. Keep in mind that this application is just a “toy” application, meaning that it might do some things more simply than a “real” API. However, I hope it will be helpful for you as you are learning the topics in this book.

## Setting up the test application

In order to use this application, you will need to do a bit of setup. There are two options to do this. You can run a version of it on GitPod, or you can run it on your local machine. I would recommend using GitPod, as that approach should have all the dependencies set up for you automatically. You can do that with the following steps:

1. First of all, you will need a GitHub account. If you don't yet have one, you can sign up at <https://github.com>.
2. Once you have that account, navigate to <https://gitpod.io/#https://github.com/djwester/todo-list-testing> in your browser.
3. Click to **Continue with GitHub**.
4. You can accept the default workspace settings and continue.
5. Once it has finished loading, go to the terminal, type in the command `make run-dev`, and hit *Enter*.
6. The service should start, and you should see a toast message with a few options. Click on the **Make Public** option.
7. Click on the **Ports** tab, and you can see the public URL of the test site available to copy.

Note that GitPod will shut this site down after a few minutes of inactivity, so if you haven't been using it for a while and you get an error when making API calls, you may have to come back and repeat these steps to restart the site. Also, every time you restart the site, it will have a slightly different URL, so don't forget to update the URL of any API calls pointing at this site.

If you want to instead run the site locally on your own computer, you will need to make sure you have a few pre-requisites in place. First of all, you will need to have Python version 3.11 installed:

1. Go to <https://www.python.org/downloads/>.
2. You will want to download Python 3.11, so click on the link to your platform below the **Download Python** button.

3. On the resulting page, find version 3.11 (the subversion doesn't matter) and download and install it.

You will also need to install the poetry package manager. You can do this by running the following command in the command prompt:

```
curl -sSL https://install.python-poetry.org | python3 -
```

Once you have those pre-requisites, you will need to download the site from GitHub. Make sure you have followed the instructions to install Git from <https://github.com/git-guides/install-git>. You can then clone the repository from GitHub by going to the command prompt, navigating to the folder you want to download it into, and calling this command:

```
git clone https://github.com/djwester/todo-list-testing.git
```

After it has downloaded, navigate to the folder:

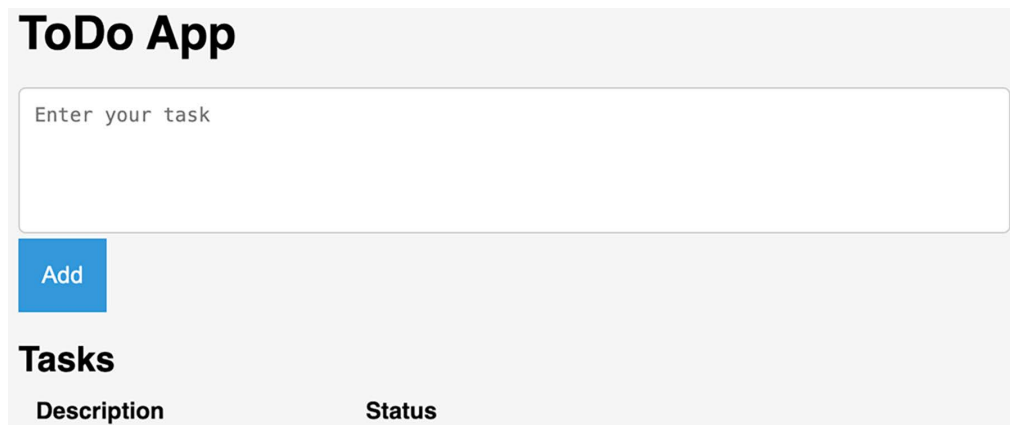
```
cd todo-list-testing
```

Start the application by calling `make run-dev`. You can now access the application by going to `https://localhost:8000`.

Now that you have this application running, let's look at sending a call to it.

## Making a call to the test application

Let's practice some of the theory we have covered so far. First, open the application in your web browser. You should see a page like this:



**ToDo App**

Enter your task

Add

**Tasks**

Description	Status
-------------	--------

Figure 1.11: Todo list application

Type a task into the box and click on Add to add it to the list. You should see it appear in the list below the box. Now that there is something on the site, let's look at accessing it through the API:

1. Create a new collection in Postman and name it `ToDoList`.
2. Add a request to the collection and name it `Get Todo List Item`.
3. In the URL field, enter the URL of the site that you built and then add `/tasks` to the end of it.
4. Send the request.

You should get back a response showing the item you created on the website.

## A challenge

I've given you a lot of step-by-step instructions on how to make API calls, but now I have a challenge for you. I want you to try to do this without me giving you the exact steps to do it.

The documentation for this todo list site says that there is an endpoint like this:

```
/tasks/{task_id}
```

Can you call that endpoint in Postman for this todo list item that you've made? Don't forget that the curly braces around `task_id` indicate that it is a request parameter.

I will use the site in a few more examples and challenges as we go through this book, but for now, feel free to play around with the API and try out some of the things that we have covered already.

## Considerations for API testing

We have started with the mechanics of how to make API requests, but this book is about API testing, so now that you know some of the basics of how an API request works, let's look at some of the things to consider when you test an API. One important aspect of testing is exploratory testing.

### Beginning with exploration

I can still clearly remember the first time I saw a modern web API in action. The company I was working at was building a centralized reporting platform for all the automated tests, and I was assigned to help test the reporting platform. One of the key aspects of this platform was the ability to read and manipulate data through a web API. As I started testing this system, I quickly realized how powerful this paradigm was.

Another part of my job at that time was to work with a custom-built test automation system. This system was quite different from the more standard test automation framework that many others in the company were using.

However, the fact that the new reporting platform had an API meant that my custom test automation system could put data into this reporting platform, even though it worked very differently from the other test automation systems. The test reporting application did not need to know anything about how my system, or any other one, worked. This was a major paradigm shift for me and was probably instrumental in leading me down the path to writing this book. However, something else I noticed as I tested this was that there were flaws and shortcomings in the API.

It can be tempting to think that all API testing needs to be done programmatically, but I would argue that the place to start is with exploration. When I tested the API for that test reporting platform, I barely knew how to use an API, let alone how to automate tests for it, and yet I found many issues that we were able to correct. If you want to improve the quality of an API, you need to understand what it does and how it works. You need to explore it.

But how do you do that?

Thankfully, Postman is one of the best tools out there for exploring APIs. With Postman, you can easily try out many different endpoints and queries, and you can get instant feedback on what is going on in the API. Postman makes it easy to play around with an API and to go from one place to another. Exploring involves following the clues that are right in front of you. As you get results back from a request, you want to think of questions that those results bring to mind and try to answer them with further calls. This is all straightforward with Postman. To illustrate, I will walk you through a case study of a short exploratory session using Postman.

For this section, I will use the `https://swapi.dev/` API. This is a fun little API that exposes data about the Star Wars movies. Don't worry if you aren't into Star Wars. No specialized knowledge is needed!

## Exploratory testing case study

Let's try this out. First, create a new collection called *Star Wars API* and add a request to it called *Get People*. Put `https://swapi.dev/api/people/1/` into the URL field and send that request. You should get back a response with some data for the character *Luke Skywalker*:

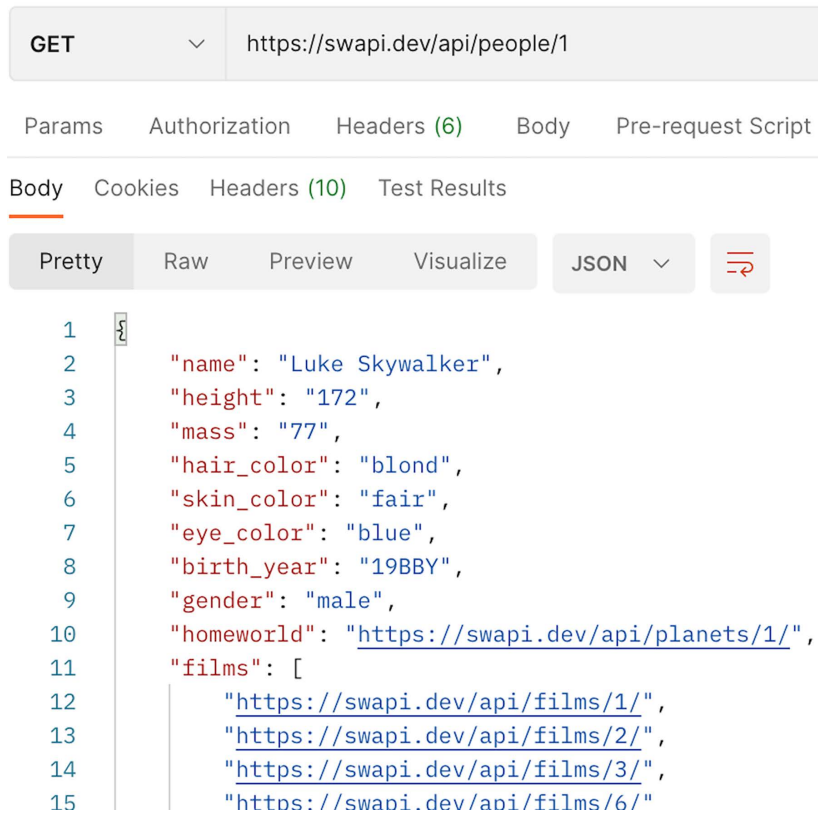


Figure 1.12: Response for the Star Wars API request

In the response, there are links to other parts of the API. A response that includes links to other relevant resources is known as a **Hypermedia** API. The first link in the films list will give us information about the film with ID 1. Since we are exploring, let's go ahead and look at that link and see what it does. You can just click on it and Postman will open a new requests tab with it already loaded. At this point, you know what to do: just click on **Send** and Postman will give you data about that first film. Under the characters in that film, you can see that there are several different characters, including, of course, a link to `/people/1`.

Seeing this triggers a thought that there might be more things to check in the /people API, so let's go back and explore that part of the API a bit more. Click on the **Get People** tab to go back to that request, and change the URL to remove the /1 from the end of it. You can now click **Send** to send a request to the endpoint, `https://swapi.dev/api/people`. This response gives back a list of the different people in the database. You can see at the top of the response that it says there is a count of 82.

We are in exploration mode, so we ask the question, "I wonder what would happen if I tried to request person number 83?" This API response seems to indicate that there are only 82 people, so perhaps something will go wrong if we try to get a person who is past the end of this dataset. In order to check this, add /83 to the end of the URL and click **Send** again. Interestingly (if the API hasn't changed since I wrote this), we get back data for a Star Wars character. It seems like either the count is wrong, or perhaps a character has been removed somewhere along the line. Probably, we have just stumbled across a bug!

Whatever the case may be, this illustrates just how powerful a little bit of exploration can be. We will get to some powerful ways to use Postman for test automation later in this book, but don't rush right past the obvious. API testing is testing. When we are testing, we are trying to find out new information or problems that we might have missed. If we rush straight to test automation, we might miss some important things. Take the time to explore and understand APIs early in the process.

Exploration is a key part of any testing, but it takes more than just trying things out in an application. Good testing also requires the ability to connect the work you are doing to business value.

## Looking for business problems

When considering API testing and design, it is important to consider the business problem that the API is solving. An API does not exist in a vacuum. It is there to help the company meet business needs. Understanding what those needs are will help to guide the testing approaches and strategies that you take. For example, if an API is only going to be used by internal consumers, the kinds of things that it needs to do and support are very different from those needed if it is going to be a public API that external clients can access.

When testing an API, look for business problems. If you can find problems that prevent the API from doing what the business needs it to do, you will be finding valuable problems and enhancing the quality of the application. Not all bugs are created equal.

## Trying weird things

Not every user of an API is going to use it in the way that the API was written for. We are all limited by our own perspectives on life, and it is hard to get into someone else's mind and see things from their perspective. We can't know every possible thing that users of our system will do, but there are strategies that can help you improve seeing things in a different light. Try doing some things that are just weird or strange. Try different inputs and see what happens. Mess around with things that seem like they shouldn't be messed with. Do things that seem weird to you. Often, when you do this, nothing will happen, but occasionally, it will trigger something interesting that you might never have thought of otherwise.

Testing does not need to be the mindless repetition of the same test cases. Use your imagination. Try strange and interesting things. See what you can learn. The whole point of this book is for you to learn how to use a new tool. The fact that you have picked up this book shows that you are interested in learning. Take that learning attitude into your testing. Try something weird and see what happens.

There is obviously a lot more to testing than just these few considerations that I have gone over here. However, these are some important foundational principles for testing. I will cover a lot of different ways to use Postman for testing in this book, but most of the things that I talk about will be examples of how to put these strategies into practice. Before moving on to more details on using Postman though, I want to give you a brief survey of some of the different types of APIs that you might encounter.

## Different types of APIs

There are several types of APIs commonly used on the internet. Before you dive too deep into the details of using Postman, it is worth knowing a bit about the different kinds of APIs and how to recognize and test them. In the following sections, I will provide brief explanations of the three most common types of APIs that you will see on the internet.

### REST APIs

We'll start with what is probably the most common type of API you'll come across on the modern web, the **RESTful API**. REST stands for **R**epresentational **S**tate **T**ransfer and refers to an architectural style that guides you in terms of how you should create APIs. I won't go into the details of the properties that a RESTful API should have (you can look them up on Wikipedia if you want, at [https://en.wikipedia.org/wiki/Representational\\_state\\_transfer](https://en.wikipedia.org/wiki/Representational_state_transfer)), but there are a few clues that can let you know that you are probably testing a RESTful API.



Since RESTful APIs are based on a set of guidelines, they do not all look the same. There is no official standard that defines the exact specifications that a response must conform to. This means that many APIs that are considered to be RESTful do not strictly follow *all* the REST guidelines. REST in general has more flexibility than a standards-based protocol such as **SOAP** (covered in the next section), but this means that there can be a lot of diversity in the way REST APIs are defined and used.

*So how do you know whether the API that you are looking at is RESTful?*

Well, in the first place, what kind of requests are typically defined? Most REST APIs have GET, POST, PUT, and DELETE calls, with perhaps a few others. Depending on the needs of the API, it may not use all these actions, but those are the common ones that you will likely see if the API you are looking at is RESTful.

Another clue is in the types of requests or responses that are allowed by the API. Often, REST APIs will use JSON data in their responses (although they could use text or even XML). Generally speaking, if the data in the responses and requests of the API is not XML, there is a good chance you are dealing with a REST-based API of some sort. There are many examples of REST APIs on the web, and in fact, all the APIs that we have looked at so far in this book have all been RESTful.

## SOAP APIs

Before REST, there was **SOAP**. SOAP has been around since long before Roy Fielding came up with the concept of REST APIs. It is not as widely used on the web now (especially for smaller applications), but for many years, it was the default way to make APIs, so there are still many SOAP APIs around.

SOAP is an actual protocol with a **W3C** standards definition. This means that its usage is much more strictly defined than REST, which is an architectural guideline as opposed to a strictly defined protocol.

If you want a little light reading, check out the w3 primer on the SOAP protocol (<https://www.w3.org/TR/soap12-part0/>). It claims to be a:



---

*non-normative document intended to provide an easily understandable tutorial on the features of SOAP Version 1.2.*

---

OK, so the reading might not be quite as light as I promised, but looking at some of the examples in there may help you understand why REST APIs have become so popular. SOAP APIs require a highly structured XML message to be sent with the request. Being built in XML, these requests are not so human-readable and require a lot of complexity to build up. There are, of course, many tools (such as SoapUI) that can help with this, but in general, SOAP APIs tend to be a bit more complex to get started with. You need to know more information (such as the envelope structure).

*But how do you know whether the API you are looking at is a SOAP API?*

The most important rule of thumb here is, does it require you to specify structured XML in order to work? If it does, it's a SOAP API. Since these kinds of APIs are required to follow the W3C specification, they must use XML, and they must specify things such as `env:Envelope` nodes inside the XML. If the API you are looking at requires XML to be specified, and that XML includes the **Envelope** node, you are almost certainly dealing with a SOAP API.

Another indicator that you are dealing with a SOAP API is the existence of a **Web Services Description Language (WSDL)** file. This file is used to define the contract for how the API works. It is meant to be used programmatically by the consumer to help with calling the API. Not every SOAP API provides one, but many do, and if there is one, you can be sure that you are working with a SOAP API.

## SOAP API example

Let's look at an example of what it would look like to call a SOAP API. This is a little bit harder than just sending a GET request to an endpoint, but Postman can still help us out with this. For this example, I will use the country info service to get a list of continents by name. The base page for that service is here: <http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso>. In order to call this API in Postman, we will need to set up a few things. You will, of course, need to create a request in Postman. However, in this case, instead of having the request method as a GET request, you will need to set the request method to POST and then enter the URL specified above. SOAP requests are usually sent with the POST method rather than the GET method, as they have to send XML data in the body of the request. You usually don't send body data in a GET request, so most SOAP services require the requests to be sent using the POST protocol.

However, **don't** click **Send** yet. Since this is a SOAP API, we need to send some XML information as well. We want to get the list of continents by name, so if you go to the CountryInfoServices web page, you can click on the first link in the list, which will show you the XML definitions for that operation. Use the SOAP 1.2 example on that page and copy the XML for it.

In Postman, you will need to set the input body type to **raw**, choose **XML** from the dropdown, and then paste in the Envelope data that you copied from the documentation page. It should look something like this:

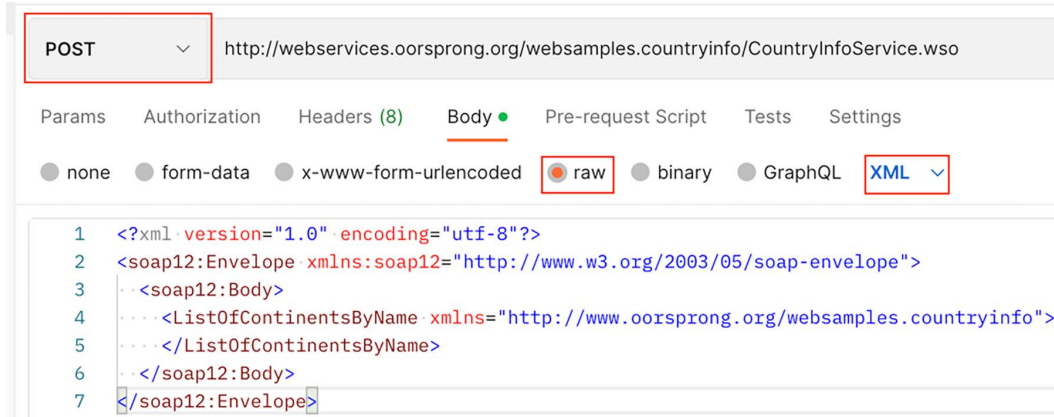


Figure 1.13: SOAP API information

For this particular API, we also need to modify the Content-Type header. Go to the **Headers** tab and click on the button labeled **hidden**. You will see that Postman has automatically added a Content-Type header with a value of `application/xml`. However, this API needs the content type set to `application/soap+xml`. We can't directly modify the automatically created header, but we can add one that replaces it. At the bottom of the list, type Content-Type into the **Key** field and set the value to `application/soap+xml`. Postman will automatically put a strikethrough in the autogenerated header, indicating that it won't use it and will instead use the one you specified. To be extra sure, you can deselect it from the list as well.

POST    http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso

Params    Authorization    **Headers (9)**    Body    Pre-request Script    Tests    Settings

Headers    Hide auto-generated headers

	KEY	VALUE
<input checked="" type="checkbox"/>	Postman-Token	<calculated when request is sent>
<input type="checkbox"/>	Content-Type	application/xml
<input checked="" type="checkbox"/>	Content-Length	<calculated when request is sent>
<input checked="" type="checkbox"/>	Host	<calculated when request is sent>
<input checked="" type="checkbox"/>	User-Agent	PostmanRuntime/7.30.1
<input checked="" type="checkbox"/>	Accept	*/*
<input checked="" type="checkbox"/>	Accept-Encoding	gzip, deflate, br
<input checked="" type="checkbox"/>	Connection	keep-alive
<input checked="" type="checkbox"/>	Content-Type	application/soap+xml

Key    Value

Figure 1.14: Content-Type header set to application/soap+xml

Now, you are finally ready to click on **Send**. You should get back a list of the continents. As you can see, there is a lot more complexity to calling SOAP APIs. This complexity is one of the contributors to the fall in popularity of these kinds of APIs, and it also indicates to us what type of API this is. REST APIs can, of course, have complex bodies specified as well, but the requirement to do this in XML and the existence of the **Envelope** node in this indicates that this API is indeed a SOAP API.

## GraphQL APIs

SOAP came before REST, and in many ways, REST was designed to deal with some of the shortcomings of SOAP. Of course, in software, we are never done making things better, so we now have a type of API known as GraphQL. GraphQL is a query language that was designed to deal with some of the situations where REST APIs have shortcomings. RESTful APIs are not aware of what specific information you might be looking for, so when you call a REST API endpoint, it gives you all the information it has. This can mean that you can receive extra information that you don't need, or it can mean that you don't receive all the information you need and that you must call multiple endpoints to get what you want. Either of these cases can slow things down, and for big applications with many users, that can become problematic. GraphQL was designed by Facebook to deal with these issues.

GraphQL is a query language for APIs, so it requires you to specify in a query what you are looking for. With REST APIs, you will usually only need to know what the different endpoints are in order to find the information you are looking for, but with a GraphQL API, a single endpoint will contain most or all of the information you need, and you will use queries to filter down that information to only the bits that you are interested in. This means that with GraphQL APIs, you will need to know the schema or structure of the data so that you know how to properly query it, instead of needing to know what all the endpoints are.

*How do you know whether the API you are looking at is a GraphQL API?*

Well, if the documentation tells you about what kinds of queries you need to write, you are almost certainly looking at a GraphQL API. In some ways, a GraphQL API is similar to a SOAP API, in that you need to tell the service some information about what you are interested in. However, a SOAP API will always use XML and follow a strict definition in the calls, whereas GraphQL APIs are usually a bit simpler and not defined in XML. Also, with GraphQL, the way the schema is defined can vary from one API to another, as it does not need to follow a strictly set standard.

## GraphQL API example

Let's look at a real-life example of calling a GraphQL API to understand it a bit better. This example will use a version of the countries API that you can find hosted at <https://countries.trevorblades.com/>. You can find information about the schema for it on GitHub: <https://github.com/trevorblades/countries>. Now, you could just create queries in the playground provided, but for this example, let's look at setting it up in Postman.

Similar to calling a SOAP API, we will need to specify the service we want and do a POST request rather than a GET request. GraphQL queries can be done with GET, but it is much easier to specify the query in the body of a POST request, so most GraphQL API calls are sent with the POST method. In fact, you will notice that when you choose the GraphQL option for the body of your request in Postman, it automatically changes the type to POST for you in anticipation that it is what a GraphQL request will need. So go ahead and do that; choose the **GraphQL** option on the **Body** tab and enter the query that you want:

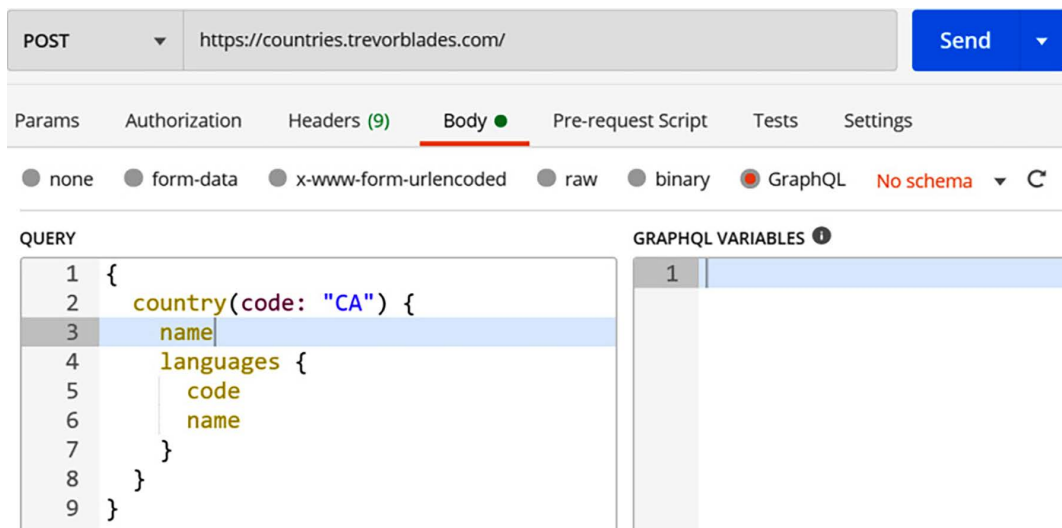


Figure 1.15: GraphQL query

As you type in your query, you might notice that Postman offers some autocomplete options. It can do this because it knows what structure the data has. Postman will automatically read the GraphQL schema for you and use that to help with constructing queries.

As you can see, in this example, I have requested the name and languages of Canada. Once I have specified this information, I can click **Send**, and I get back some JSON with the country name and a list of the official languages. If I wanted additional information (say the name of the capital city), I could just modify the query to include a request for that information and send it off again using the same endpoint, and I would get back the new set of information that I requested.

At this point, I have obviously only been able to give a very quick introduction to each of these types of APIs. If you are working with a GraphQL or SOAP API, you may need to spend a bit of time making sure you understand a little more about how they work before proceeding with this book. Most of the examples through the rest of this book will use RESTful APIs.

However, the concepts of API testing will largely stay the same, regardless of the type of API testing that you need to do. You should be able to take the things that you will learn in this book and put them to use, regardless of the type of API you work with in your day job.

## Summary

Let's pause for a minute to consider everything we have gone over in this chapter. You've installed Postman and already made several API requests. You've learned how API requests work and how to call them. I explained some basic testing considerations and gave you strategies that you can start to use right away in your day-to-day work. You also got to make calls to GraphQL, SOAP, and REST APIs and learned a ton of API terminology.

You now have something firm to hold onto as we proceed through the rest of this book. I will take you deep into a lot of API testing and design topics and help you get the most out of Postman, but in order to get the most out of it and not feel frustrated, it would be good to make sure you understand the topics covered in this chapter.

Take a minute to ask yourself the following questions:

- Would I feel comfortable reading an article on API testing? Could I follow along with the terminology used?
- What are some basic strategies and approaches that I can use in API testing?
- If I was given an API endpoint, could I send a request to it in Postman? What things would I need to do to send that request?
- If I was given some API documentation, could I figure out what kind of API it was and send requests to it?

If you can answer these questions, you certainly have the grounding that you need to move on in this book. If you are not totally sure about some of them, you might want to review some of the relevant sections in this chapter and make sure you have a solid grip on them.

You've learned a lot already! In the next chapter, we will dive into some of the principles of API design and look at how to use Postman to help put those principles into practice when creating an API.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.com/invite/nEN6EBYPq9>







# 2

## API Documentation and Design

I believe that design, testing, and documentation are just as much a part of creating a quality product as development is. Perhaps they are even more important. You might find it surprising that a book on testing is spending so much time talking about documentation and design, but both of these practices are also key contributors to quality.

Good design can make the life of a developer or tester much easier. Designing something isn't so much about making it look good as it is about ensuring that it brings satisfaction. We, humans, enjoy things that look nice, so designs can sometimes be focused on the look and feel of something, but even in the realm of APIs, where there isn't much to "see," good design can make it much more enjoyable to use and thus improve the quality.

Good documentation also makes the life of a developer and tester much easier. I have struggled through working with APIs that have missing or incorrect API documentation. It makes it much harder to work with and test the API. You can spend hours trying to figure out how to make one call simply because there is one missing parameter that you need to include for the call to work, but with good documentation, you could have figured that out in minutes.

This chapter will go over the principles of API documentation and design and cover the following main topics:

- Figuring out the purpose of an API
- Creating usable APIs
- Documenting your API (with Postman)
- API design example (a practical exercise)

The material in this chapter can feel a bit abstract and theoretical, but I have included a few exercises to help you figure out how to use these concepts in practice. I would encourage you to spend the time to work through those exercises and, by the end of this chapter, you will be able to use these design principles to come up with great insights for APIs that you are currently working on, as well as having a strong foundation that you can use if you need to create a new API. This chapter will also help you get started with API documentation in Postman and show you how to use the RAML specification language to design and model an API and to automatically add requests into Postman.

## Technical requirements

The code used in this chapter can be found at <https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/master/Chapter02>.

## Start with the purpose

We don't just build an API because it is a fun thing to do. We build APIs to achieve a purpose. They help us or our users solve a problem. This might seem obvious, but let's be honest: we forget this far too often.

Design is a difficult domain. Perhaps your company has designers (as mine does). Technical people such as testers and developers can sometimes dismiss what designers do as “just making things look pretty,” but the reality is that good design is very hard. Good design makes something that is suited for its purpose. This book is not a book on designs (if you want to read more about the theory of design, check out books such as *The Design of Everyday Things* by Don Norman). However, if you are interested in good-quality APIs, you need to think for a few minutes about the purpose of your API and how you will design it to meet that purpose.

Knowing that you need to do something is an important first step. Unfortunately, many talks, articles, and books stop there. But what good does that do you? You are convinced that it makes sense to design an API with the purpose in mind, but what is the purpose? What do you do if you don't know? How do you figure out the purpose of your API?

## Figuring out the purpose of an API

You want to figure out the purpose of your API, but how do you do that? There are a few simple strategies that you can use for this. The first thing is to ask questions. If you or your team is writing an API, you were probably asked to do it by someone. Talk to that person! Why do they want the API? What do they want to do with it? What problems do they think it will solve? Ask them these questions and see if you can figure out some of what the purpose of the API is.

### Personas

Another simple strategy that you can use is personas. A persona is simply a made-up person that you use to help you think through who might be using your API. For example, you might have a persona representing a user in your company who will be using the API to develop user interfaces, or you might have a persona of a software engineer who works for one of your clients and is using the API to develop a custom application. Thinking through different kinds of users and considering how they might interact with your API will help you understand the purpose that it serves.

When creating a persona, think about things that go beyond just the technology that the person might use or understand. Think about things like the goals and motivations they might have and the things that might frustrate them. It's also helpful to think of some personal details that might make them a bit more relatable to you. For example, you could think about whether they are a dog person or whether they have kids. Is there some other thing that makes them unique? In other words, what kind of person are they? Writing down details like this can seem a bit silly at times, but it helps you better empathize with this persona, and as you are able to do that, you will be more able to put yourself in their shoes and understand the kinds of things that are important to them. The more you understand this, the better you will be able to understand the purpose of your API.

### The why

At the heart of figuring out the purpose is the question of why. Why are we making this API? The why is almost always about solving a problem. A great way to figure out the purpose of an API is to figure out what problem it solves. Does it make it easier to write UI elements? Does it enable clients to customize the application? Does it enable third-party developers to use your platform? Does it simplify integrations with other applications? What problem is your API solving? Answer these questions and you will be well on your way to knowing the purpose of your API.

**NOTE:**

This exercise of figuring out the purpose of an API doesn't just apply to new APIs. If you are working with an existing API, there is a lot of value in understanding the purpose. It may be too late to radically alter the design of the API, but there is no more important threat to quality than not actually helping people solve the problems they need to solve. If nothing else, understanding the purpose of an existing API that you are testing will help you figure out which bugs are important and which ones might not matter as much. It takes some skill to figure out which bugs are urgent and which are not, and understanding the purpose of the API helps with that.

**Try it out**

I have just given you some practical advice, but it won't do you much good if you don't use it. This book isn't just about filling your head with theory. It is about helping you get better at testing APIs. A little later in this book, I will show you some of the tools that Postman has for helping with API design, but right now, I want you to pause and try out what we have just been talking about.

Take an existing API that you have been working on and see if you can write down the purpose in two or three sentences. Use the following steps to work through the process:

1. Identify at least two key stakeholders for the API. Do this by asking the question "Who wants (or wanted) this API built?" Write down these stakeholder names.
2. If possible, talk to those stakeholders and ask them what they think the purpose of this API should be and why they want to build it. Write down their answers.
3. Create preferably two (but at least one) personas that list out the kinds of people that you think will be using the API. What skill level do they have? What work are they trying to accomplish? How will your API help them?
4. Write down what problem(s) you think the API will solve.
5. Now, take all the information that you have gathered and look through it. Distill it down into two or three sentences that explain the purpose of the API.

Once you've completed this exercise, you will have a good grasp of the purpose of the API you are investigating. The result of that might look like something like this:

"This API was built so that other services in our company could communicate with our service. It will be primarily consumed by other developers in the company. It is an internal API that will only be accessible inside of our internal company network.

Its purpose is to let other parts of our system know about the status of certain events in our service and so some of the endpoints will be polled intermittently, but overall, it isn't expected to have heavy usage rates."

It is important to understand the purpose of an API; however, to fulfill that purpose, an API needs to be usable. Let's move on to look at what makes a usable API.

## Creating usable APIs

Usability is about the balance between exposing too many controls and too few. This is a very tricky thing to get right. On the extremes, it is obvious when things are out of balance. For example, the Metropolitan Museum of Art has an API that gives you information about various art objects in their possession. If all the API did was provide one call that gave you back all that data, it would be providing too few controls. You would need to do so much work after getting the information that you might as well not use the API at all. However, if, on the other hand, the API gave you a separate endpoint for every piece of metadata in the system, you would have trouble finding the endpoint that gave you the particular information you wanted. You would need to comprehend too much in order to use the system.

You need to think carefully about this if you want to get the balance right. Make sure your API is providing users with specific enough data for the things they need (this is where knowing the purpose comes in handy) without overwhelming them. In other words, keep it as simple as possible.

## Usable API structure

One thing that can help create a usable API is to use only **nouns** as endpoints. If you want users to be able to understand your API, structure it according to the objects in your system. For example, if you want to let API users get information about the students in a learning system, don't create an endpoint called `/getAllStudents`. Create one called `/students` and call it with the GET method. One of the reasons for doing this has to do with creating new students. If your endpoint is called `/students`, you can easily call it with a POST method to create a new student. However, if you had named it `/getAllStudents`, it would feel very unnatural to call that with a method to create new items. You can think about it in terms of saying it out loud. Saying "I want to create a new student object" sounds natural, but saying "I want to create a get all student object" does not.

Creating endpoints based on nouns will also help you more naturally structure your data. For example, if you have `/students` as an endpoint, you can easily add an endpoint for each student at `/students/{studentId}`. This kind of categorization structure is another helpful API design principle to keep in mind.

Creating a structure like this maps the layout of the API to the kinds of things that the API user needs information about. This makes it much easier to know where to find the relevant information.

A structure like this works nicely, but does it really match up with how users will interact with the API? If I am looking for information about a student, am I going to know what their ID is in the API? Perhaps, but more likely I will know something like their name. So, should we modify the structure to have an additional endpoint like `/students/name`? But what if we are looking at all the students of a certain age? Should we add another endpoint, `/students/age`? You can see where I am going with this. It can get messy quickly.

This is where **query parameters** are helpful. A query parameter is a way of getting some subset of the category based on a property that it has. So, in the examples that I gave earlier, instead of making “name” and “age” endpoints under the “students” category, we could just create query parameters. We would call `/students?name='JimJones'` or `/students?age=25`. Query parameters help keep the endpoints simple and logical but still give the users the flexibility to get the information they are interested in effectively.

## Good error messages

A usable API helps the users when they make mistakes. This means that you give them the correct **HTTP codes** when responding to a call. If the request is badly formatted, the API should return a 400 error code. I won't list all the HTTP codes here as they are easily searchable online, but ensuring that your API is returning codes that make sense for the kind of response you are getting is an important quality consideration.

In addition to the HTTP codes, some APIs may return messages that let you know what possible steps you can take to correct this issue. This can be very helpful, although you will want to be careful that you don't reveal too much information to those who might be bad actors.

## Documenting your API

One often-overlooked yet vitally important aspect of a good-quality API is proper documentation. Sometimes, testers and developers can overlook documentation as something that is outside of their area of expertise. If you have a team that writes documentation for your API, it is certainly fine to have them write it, but don't treat documentation as a second-class citizen! Documentation is often the first exposure people have to your API, and if it does not point people in the direction they need to go, they will get confused and frustrated. No matter how well you have designed your API, users will need some documentation to help them know what it can do and how to use it.

## Documenting with Postman

Postman allows you to create documentation directly in the application. In *Chapter 1, API Terminology and Types*, I showed you how to create a request to the GitHub API. Of course, GitHub has its own API documentation, but let's look at how you can create documentation in Postman using that API request. If you did not create the GitHub collection, you can import the collection from the GitHub repo for this chapter:

1. Download the GitHub API Requests.postman\_collection.json collection file from <https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/master/Chapter02>.
2. In Postman, click on the **Import** button at the top of the navigation tree:

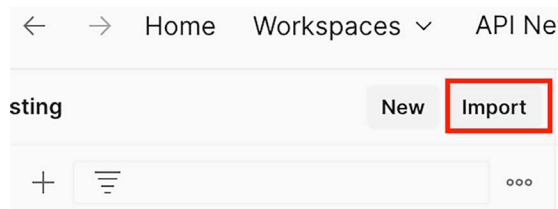


Figure 2.1: The Import button

3. On the resulting dialog, click on the **Choose Files** button, browse to where you downloaded the collection file, and select it.
4. Click on the **Import** button and Postman will import the collection for you.

Once you have the collection set up, you can create some documentation with the following steps:

1. The first thing to do is navigate to the **GitHub API Requests** collection and click on it.
2. Click on the **View complete documentation** link.

This brings up the **Documentation** panel.

In the top section, you can enter documentation for the collection itself, if you want. There might be times when you want to do this but, in this case, you only have one request so there probably isn't too much documentation to put in here. Instead, you can scroll down to the next section to enter documentation for the **Get User Repos** request.



Note that you can also jump to a request by clicking on it in the right-hand panel, as shown in the following figure:

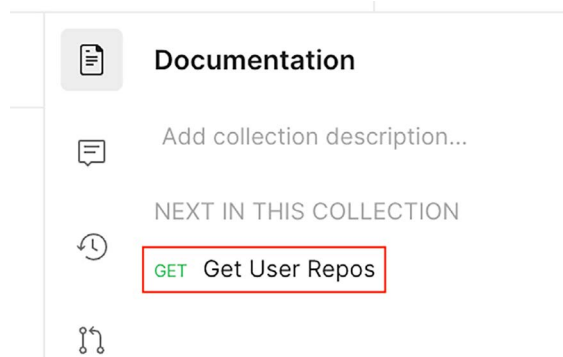


Figure 2.2: Going to the request documentation

In this case, we currently only have one request in this collection, so it is just as easy to scroll. However, if you have many requests, the jump navigation will let you quickly navigate to the one you are interested in. In order to edit the documentation, simply click on the area below the URL for the request and you can start typing your documentation for this request directly into the provided textbox.

**NOTE:**



Postman API descriptions support Markdown. If you are not familiar with Markdown, it is a simple set of rules that can be used to format text. A great resource showing the kinds of commands available is the Markdown cheat sheet, which you can find at <https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet>.

Let's write down some documentation for this endpoint:

1. At this point, just type something simple in here so that you can try it out. Type in something like this: This endpoint will get information about what repos the given user has.
2. When you click away from the editing panel, Postman will auto-save your changes. However, this documentation is currently only available in this collection. You can make it more public by publishing it.

3. In order to publish, click on the **Publish** icon near the top of the page, as shown in the following figure:

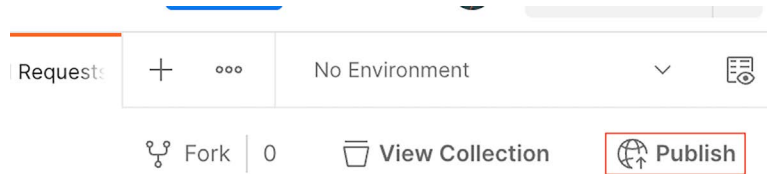


Figure 2.3: Publish documentation

4. This opens a web page where you can choose various styling options for your documentation. Once you have set up the options that you want on this page, you can scroll down and click on the **Publish Collection** button to make this documentation available to others.
5. The published page tells you the URL from which you can see the documentation that you made. Click on that and you will see a nice documentation page that you can share with anyone.

When looking at this documentation, it seems a bit sparse. It could be improved with an example request showing what a response looks like. You can easily add examples in Postman with the following steps:

1. Return to Postman and go to the `Get User Repos` request.
2. Replace the `{{username}}` variable in the request URL with a valid username (you could use your own or just put in mine, which is `djwester`). In future chapters, I will show you how to work with variables, but for now, just manually replace it.
3. Click on the **Send** button for the request to send it off. If you get back an empty array, it could be that your GitHub has privacy settings that don't allow these results to be returned. In that case, use my username (`djwester`) instead.
4. Now go down to the response and click on the **Save as example** button.



Figure 2.4: Adding examples

5. Postman will automatically add an example to this request. Return to your API documentation page and refresh it. You will now see an example request and response:

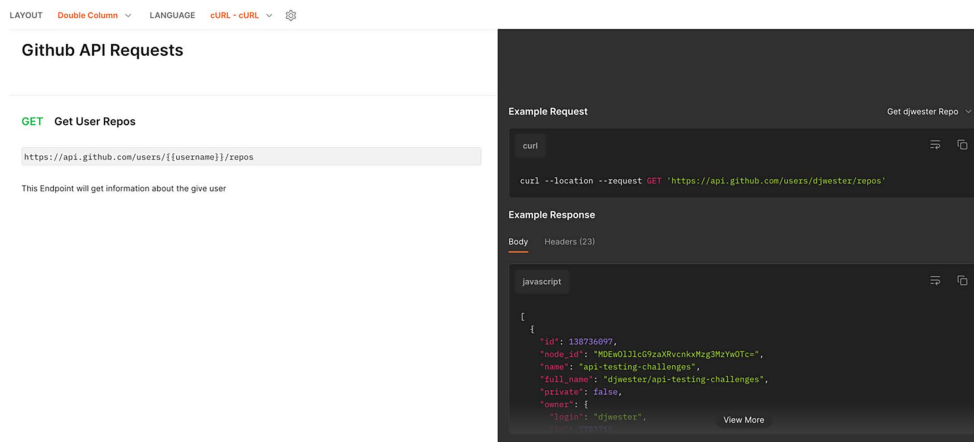


Figure 2.5: Example API request and response in documentation

It is worth spending more time exploring how to document your API well, but these features make it straightforward to do in Postman. Whether you are a tester or developer and whether you are working on a new API or one that has been around for some time, I would encourage you to take the time to properly document your API. Good documentation will make a world of difference to your users and will also lead to you and your team having a better understanding of how it works. It is well worth the time that it takes, especially when working with a tool like Postman that makes it so easy to do. In fact, although this is not a book on API documentation, it is worth spending a few minutes on some of the theory of how to write good API documentation.

## Good practices for API documentation

One key practice to keep in mind when writing API documentation is **consistency**. In this chapter and the previous one, I have been giving you a lot of API-related terminology. There is always going to be some variation in how terms are used but, as much as possible, try to use terms in a way that is consistent with how other APIs use them. You should also try to have as much internal consistency as possible in your documentation. This means things like using the same terms to mean the same thing throughout the documentation. It also means that, as much as possible, you want your API structure to be consistent so that you don't have to explain similar things in different ways throughout your documentation.

Writing API documentation can benefit you and not just those reading it. Sometimes, in reading or creating documentation, things will jump out at you. You might see how two things that are quite similar are laid out in very different ways. This can help you see where you might need to fix or change things in the API itself, which is another benefit of good API documentation.

Another very important thing to keep in mind when it comes to API documentation is the importance of examples. We already saw how easy it is to make examples in Postman documentation. Take advantage of that! It has been said that a picture is worth a thousand words. I think that an example in an API doc is worth nearly as many. Some things are difficult to explain well in words and it takes seeing it done in an example for people to grasp how it works. Do not underestimate the power of a well-crafted example.

One final thing to keep in mind with API documentation is a problem that happens with all documentation. It gets out of date. Code changes, and that applies to API code as well. Over time, what your API does and the exact ways that it does it will change. If you don't keep your documentation up to date to match those changes, the documentation will soon cease to be useful.

You have already seen how you can keep your documentation right with your tests and how you can generate examples directly from your requests in Postman. Take advantage of that and make sure your documentation stays up to date. Postman will take care of a lot of the work for you by updating all the documentation automatically every time you publish.

There are some specification tools that can help automatically generate documentation. These kinds of tools can help keep documentation and tests up to date with the latest code changes. I will take you through some of those in more detail in *Chapter 4, Considerations for Good API Test Automation*. In that chapter, I will particularly focus on the Open API specification, as it is a powerful and popular API specification tool, but there are other API specification tools that we can use.

There is an ongoing debate in the API development community that mirrors broader debates in the software development community as a whole. The debate boils down to how much time you should spend upfront on design. Some will argue that since the only good software is software that helps someone solve a problem, we need a “ship first” mentality that gets our ideas out there. We then fill out the design based on the kinds of things clients actually need and care about. Others will argue for a “design first” approach, where you rigorously define the API behavior before you write any code. As with most things, you are probably best off avoiding either ditch and finding the middle ground between them.

Modern tooling can help us do this. For example, there are tools that will allow you to create simple designs and then use those designs to get feedback from clients. One tool that is helpful for API design is **specification languages**. These are sets of rules that can be used to define the way an API is expected to work. As we look into how to design APIs, let's look at a specification language that is meant to help shift the focus away from merely documenting an API and toward helping with API design.

## RESTful API Modeling Language

RAML, which stands for **RESTful API Modeling Language**, is an API specification language that, as the name implies, helps with modeling APIs. You can read more about it on the RAML website, <https://raml.org/>. RAML has some tools that can help with API design but getting into those is beyond the scope of what I want to cover in this book. For now, I just want to introduce you to this specification and let you see how you can use it to design an API that meets the design criterion I've talked about in this chapter.

Getting started with RAML is as easy as opening a text editor and typing in some text. RAML is meant to be human-readable and so the specification is written in a simple text-based format. RAML is also structured hierarchically, which makes it easy to create the kind of usable API structures that I've talked about. In the next section, I will walk you through an example of using this modeling language to design an API and leverage the power of that design in Postman. You will then get to try it out on your own as well!

## API design example

I have talked about a lot of the theory of API design, so now I want to look at how you can use Postman to help you out with practically designing an API. API design does not only apply to new APIs that you create. In fact, using the principles of API design when testing an existing API is a great way to find potential threats to the value of that API, but for the sake of understanding this better, let's look at how you can design an API from scratch. If you understand the principles through this kind of example, you should be able to use them on existing APIs as well.

## Case study – Designing an e-commerce API

Let's imagine that we want to design an API for a very simple e-commerce application. This application has a few products that you can look at. It also allows users to create a profile that they can use when adding items to their cart and purchasing them. The purpose of this API is to expose the data in a way that can be used by both the web and mobile application user interfaces. Your team has just been given this information and you need to come up with an API that will do this.

So, let's walk through this and see how to apply the design principles we've covered. I will start with a simple RAML definition of the API. The first thing we need is to create a file and tell it what version of RAML we are using. I did this by creating a text file in Visual Studio Code (you can use whatever text editor you prefer) called `E-Commerce_API-Design.raml`. I then added a reference to the top of the file to let it know that I want to use the 1.0 version of the RAML specification:

```
#%RAML 1.0
---
```

I also needed to give the API a title and set up the base URI for this API, so, next, I defined those in the file:

```
title: E-Commerce API
baseUri: https://api.ecommerce.com/{version}
version: v1
```

## Defining the endpoints

This is a made-up API so that base URI reference does not point to a real website. Notice also how the version has been specified. Now that I have defined the root or base of the API, I can start to design the actual structure and commands that this API will have. I need to start with the purpose of this API, which is to enable both a website and a mobile app. For this case study, I am not going to dive too deep into things like creating personas. However, we do know that this API will be used by the frontend developers to enable what they can show to the users. With a bit of thought, we can assume that they will need to be able to get product information that they can show the users. They will also need to be able to access a user's account data and allow users to create or modify that data. Finally, they will need to be able to add and remove items from the cart.

With that information in hand about the purpose of the API, I can start to think about the usability and structure of this API. The frontend developers will probably need a `/products` endpoint to enable the display of product information. They will also need a `/users` endpoint for reading and maintaining the user data and a `/carts` endpoint that will allow the developers to add and remove items from a cart.

These endpoints are not the only way that you could lay out this API. For example, you could fold the **carts** endpoint into the **users** one. Each cart needs to belong to a user, so you could choose to have the cart be a property of the user if you wanted. It is exactly because there are different ways to lay out an API that we need to consider things like the purpose of the API. In this case, we know that the workflow will require adding and removing items from a cart regularly.

Developers will be thinking about what they need to do in those terms, and so to make them call a “users” endpoint to modify a cart would cause extra data to be returned that they do not need in that context and could also cause some confusion.

Now that I have picked the endpoints I want to use in this API, I will put them into the RAML specification file. That is simply a matter of typing them into the file with a colon at the end of each one:

```
/products:  
/users:  
/carts:
```

## Defining the actions

Of course, we need to be able to do something with these endpoints. What actions do you think each of these endpoints should have? Take a second to think about what actions you would use for each of these endpoints.

My initial thought was that we should have the following actions for each endpoint:

```
/products:  
  get:  
/users:  
  get:  
  post:  
  put:  
/carts:  
  get:  
  post:  
  put:
```

Think about this for a minute, though. If I only have one endpoint, `/carts`, for getting information about the carts, I need to get and update information about every cart in the system every time I want to do something with any cart in the system. I need to take a step back here and define this a little better. The endpoints are plural here and should represent collections or lists of objects. I need some way to interact with individual objects in each of these categories:

```
/products:  
  get:  
  /{productId}:  
    get:
```

```
/users:
  post:
  get:
  /{username}:
    get:
    put:
/carts:
  post:
  get:
  /{cartId}:
    get:
    put:
```

Here, I have defined **URI parameters** that enable users to get information about a particular product, user, or cart. You will notice that the POST commands stay with the collection endpoint, as sending a POST action to a collection will add a new object to that collection. I am also allowing API users to get a full list of each of the collections as well if they want.

## Adding query parameters

In *Chapter 1, API Terminology and Types*, you learned about query parameters. Looking at this from the perspective of the users of the API, I think it would be helpful to use a query parameter in the carts endpoint. When a user clicks on a product and wants to add that product to their cart, the developer will already know the product ID based on the item the user clicked. However, the developer might not have information about the cart ID. In this case, they would have to do some sort of search through the carts collection to find the cart that belonged to the current user. I can make that easier for them by creating a query parameter. Once again, I am using the design principles of usability and purpose to help create a good model of how this API should work.

In RAML, I just need to create a query parameter entry under the action that I want to have a query parameter:

```
/carts:
  post:
  get:
    queryParameter:
      username:
  /{cartId}:
    get:
    put:
```



I have designed the structure of the API using the principles I laid out, and hopefully, you can see how powerful these principles are in helping you narrow down a broad space into something manageable and useful. When it comes to creating a RAML specification for an API, you would still need to specify the individual attributes or properties of each of the endpoints and query parameters that you want to create. I won't go into all those details here. You can look at the RAML tutorials (<https://raml.org/developers/raml-100-tutorial>) to learn more about how to do this. I didn't give enough information about this imaginary API that we are designing to fill out all the properties. We don't know what information each product or user has, for example. In real life, you would probably get this information based on what is in the database and then build out the examples and attributes based on that.

## Using the RAML specification in Postman

This might not be a fully-fledged API specification, but it is enough for us to use in Postman! Click on the **Import** button in Postman and browse to the `.raml` file from this case study. Before importing it, check the **View Import Settings** section:

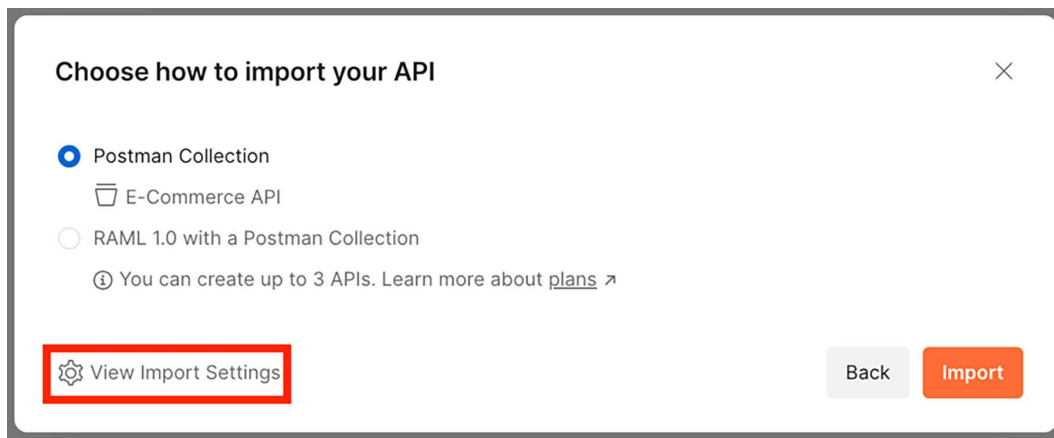


Figure 2.6: Import Settings

Ensure that both the request and response parameter generation options are set to use **Schema**. Return to the Import panel and click on **Import**. Postman will parse the specification for you and automatically create a collection that has calls defined for each of the endpoints and action combinations that you created in the specification file. Pretty cool, isn't it?

Once you have learned about some more Postman features, I will show you how to use these concepts to dive a lot deeper into the design process for APIs. For now, though, I want you to be able to think about how API design intersects with the quality of the API. In fact, I want to give you a little challenge to try out.

## Modeling an existing API design

You have been learning how to think through API design. I gave you some principles for this and then walked you through putting the principles into practice in a case study. The case study we just went through involved designing a new API from scratch. Often, though, you will be working with existing APIs. These same design principles can be used to help you find places to improve your existing APIs. In order to learn these principles, you should try this exercise out for yourself. See if you can use these principles on an API that you are currently working on. If the company you are working at does not have an API that you can use, you can, of course, just find a public API to try out this exercise with.

Using the API you have selected, work through the following steps to apply the principles of API design:

1. Add each of the endpoints to a RAML file. Make sure to follow the hierarchy of the API in that file.
2. Spend a bit of time thinking about what the purpose of the API is and reflecting on whether the structure that you see here fits with that purpose. In what other ways might you design the API? Could you improve the layout to better fit the purpose?
3. If you were designing this API, what actions and query parameters would you give to each endpoint? Create a copy of the file and fill it in with what you think you would do with this API.
4. In the original file, add the actual actions and query parameters that the API has. How do they compare to the ones that you made in the copy of the file?

If you want, you can import the file into Postman and, as you learn more about testing and other features that you can use in Postman, you will already have a collection of requests ready to use.

## Summary

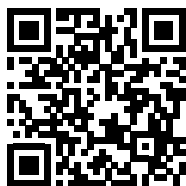
Designing an API takes careful thought. An API is software, and the whole reason we write software is to help people solve problems. APIs need to solve a problem, and the better they solve that problem, the better quality they are. One thing that can help with API quality is to have a well-designed API. A well-designed API is one that is designed to fulfill the purpose for which it is used. In this chapter, you learned how to think through the purpose of an API by coming up with personas. You also learned some questions that you can ask to get to the heart of why an API needs to exist in the first place.

This chapter also showed you some ways to structure and document APIs. You learned about API specification and how you can use RAML to create design-driven specifications for an API. And, of course, you also got to try these things out and put them into practice! With a firm grasp of the principles of API design, you are ready to dig a little deeper into API specification languages and how to use them in creating good-quality APIs. We will dive into that in the next chapter.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.com/invite/nEN6EBYPq9>



# 3

## OpenAPI and API Specifications

The history of computing has been a long path toward higher and higher levels of abstraction. In the early days of the computer era, computers were programmed with punch cards and assembly languages. The invention of FORTRAN in the 1950s created a programming language that allowed programmers to write code using commands that more closely resembled human language. Over the years, object-oriented languages such as C++ came along and added additional levels of abstraction. In a sense, APIs are another level of abstraction. They allow people to write “code” at a very high level that will tell computers what to do. However, we didn’t stop there. As APIs became more popular, we started to develop interfaces that would allow us to specify how an API works, so that users who know nothing about the source code can easily interact with the remote service.

We call these interfaces **API specification languages**. These languages help with API development in several ways. They make it easier to share information between clients and those creating the API, and they also enable a lot of automation. They let you do things such as automatically create some kinds of tests and code. There is a lot of complexity that goes into using these specifications but, when you understand them, you will find that they help you create and maintain better-quality APIs. SOAP APIs already require you to build in specifications through the XML that you need to use, and GraphQL APIs also require you to define the structure up front, and so API specification languages are generally only used for RESTful APIs.

There are several different RESTful API specification languages that have been created. In *Chapter 2, API Documentation and Design*, I talked about RAML and how it can be used to create design-driven APIs. RAML is an example of an API specification language.

Another API specification language is the **OpenAPI Specification (OAS)**. This is the most used API specification language and much of this chapter is devoted to understanding how to use it. However, before diving into how to use it, I want to spend a bit of time talking about the value of API specifications in general.

The following topics will be covered in this chapter:

- What are API specifications?
- Creating an OAS
- Using API specifications in Postman

## Technical requirements

The code used in this chapter can be found at <https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/master/Chapter03>.

This chapter will also use the online Swagger Editor tool. This is a simple tool that you don't need to download or install.

## What are API specifications?

Although SOAP APIs are often cumbersome and difficult to understand, they do have some benefits. One of the benefits of SOAP APIs is that the structure must be specified following strict programmatic rules. When a SOAP API creates a WSDL file to describe the API capabilities, the fact that it is required to follow this strict set of rules means that it is written in a way that computers can easily understand. This makes it easy to create a lot of general-purpose automation for these kinds of APIs. If a computer can understand the layout of an API, you can automatically generate some kinds of documentation, tests, and even code from that specification.

However, RESTful APIs are a bit different. They follow an architectural style that was laid out by Roy Fielding in his doctoral dissertation. This means that there are general principles that they follow, but there is not a strict protocol that they must adhere to. This balance between structure and flexibility has been a powerful concept and has contributed to the widespread adoption of this kind of API architecture. There is no such thing as a perfect solution, however, and this is no different.

Without that strict specification, each RESTful API may have some nuances to it that are different from others. As humans, this is generally fine to manage. We can quickly and easily figure out where those differences are and accommodate them. Computers struggle with this, however. Things must be very clearly and explicitly laid out for a computer to use.

This challenge with RESTful APIs was recognized early on, and so some API specification formats were proposed.

API specifications provide a structured way to define an API. If you follow the rules of the specification, you can interact with the API at a higher level of abstraction. For example, an API specification could be used to automatically create mock servers that you can use for testing and experimentation during development. You can also do things such as automatically generate documentation and contract tests. You can even use these specifications to generate some kinds of code, for both server implementations and client-side code that calls the API.

## API specification terminology

There are a few different terms that are worth understanding when it comes to using API specifications. I've been talking about API specification languages. Another term for this is an **API description format**. You can follow the API description format to create an **API description**, which is the actual metadata that describes what the API contains and can do. There are multiple API description formats, and there are also tools that will allow you to take an API description that is written in one format and convert it to another.

I have been using the term **API specification**, which is a kind of umbrella term that covers all the different types of description formats and documents. It is used a bit fluidly in the industry (as most terms are), but I will use it in this way as a term that means both the different formats or languages and the **API description documents**. API description documents are documents that contain a description of a particular API written with a certain API description format. I will usually use the more generic term “API specification” when talking about these documents, though.

A common term you will hear when working with API specifications is **schema**. This is another term that we should take a moment to understand the meaning of.

## Defining API schema

At its most basic, a **schema** is just a plan, usually in the form of a model or theory. Schemas are common in software development. For example, databases have schemas that show what columns and datatypes each table has and how the various tables are connected to each other. In a similar way, an API can have a schema. An API schema shows how the data should be structured. It will specify what types of values a response should contain and what data you need to include in the body of **POST** or **PUT** requests.

An API schema lays out a set of rules that we expect the API to follow. It will set in place rules like “This input should always be an integer” or “This input is required, and that one is optional.”

Setting up a schema with these rules allows you to automatically check if the API is following those rules. It is powerful both as a testing tool and as a design tool. Creating a schema forces you to think carefully about what kind of rules your API needs to follow and having a schema allows you to quickly create some simple tests.

As you can see, there is some complexity with using API specification languages, so let's take a minute to review a few of the main ones before diving into how to create and use an OAS in Postman.

## Types of API specifications

There are three main RESTful API specification languages: **RAML** (<https://raml.org/>), **API Blueprint** (<https://apibuildprint.org/>), and **OpenAPI** (<https://github.com/OAI/OpenAPI-Specification>). Previously, OpenAPI was called **Swagger**, so if you hear anyone talking about Swagger, just realize that it is the same thing as OpenAPI. I will be talking extensively about OpenAPI/Swagger later in this chapter, and I've already talked a bit about RAML, but let's take a quick look at a comparison between these different specifications.

### RAML

I already showed you an example of how to use the **RESTful API Modeling Language (RAML)** to define how an API works in *Chapter 2, API Documentation and Design*. This specification language uses the YAML format. YAML is a human-readable file format that makes it easy to structure data by following a few simple rules. The rules for structuring YAML are in the YAML specification (<https://yaml.org/spec/1.2/spec.html>). Since YAML is designed to be human-readable, it is quite easy to get started with and helps make RAML intuitive to understand. Much of the structure is built in a way that matches how you should think about and design APIs. This makes RAML a good choice if you are trying to create “design-first” APIs where you plan out the API ahead of time and then create an API that matches that design.

RAML is supported by Postman and is probably the second most popular API specification language. Although not as popular as OpenAPI, it has broad community support as well, so if you are looking for a design-first approach, it may be a good choice.

### API Blueprint

The API Blueprint specification uses **Markdown** as the format. You can read more about Markdown here: <https://www.markdownguide.org/cheat-sheet/>. Using Markdown makes specifications written in this format more readable. In fact, it reads almost exactly like consumer-facing documentation. The API Blueprint format is also very easy to understand and get started with. However, it isn't as widely supported by the community.

If you want to use it in Postman, you have to first use a tool to convert it into a format that Postman can use.

The API Blueprint format is integrated into some tool stacks, such as Apiary (<https://apiary.io>), but if you are working with Postman, it is probably better to choose a different API specification language if you can.

## OpenAPI/Swagger (OAS)

The most used API specification language is OpenAPI. Since it was originally called Swagger, many of the tools that support the OAS are called Swagger tools, while the specification itself is called OpenAPI. This specification language is flexible and powerful and has a lot of tooling to support it. You can directly import descriptions written in this format into Postman, and you can use a lot of other tools to automate things related to it as well. This large toolset and broad adoption have in some ways made it the de facto standard for API specifications. If you aren't sure which specification you want to use, it's probably best to just default to using OpenAPI.

There are some other niche API specification languages out there but, for the remainder of this chapter, I want to focus on how to use OpenAPI. The OAS is the most popular specification, and if you do a lot of API testing, you will probably come across an OAS file at some point. In the next section, I'll show you how to create an OAS.

## Creating an OAS

I won't spend a lot of time on the Swagger tools that are used for OpenAPI specifications, but I do want to give you at least an introduction to the subject so that you can use these tools in your API testing if you want. In this section, you will learn how to create and edit these files in the Swagger Editor. You will also learn what an API schema is and how to codify one in an API specification. To demonstrate these things, I will use the Swagger Petstore API. This is an example API that already has the OAS built for it. It is a good way to get started with using OpenAPI and is used in its documentation a lot as well.

The OAS enables a lot of value and has a lot of available features. In this section, I will quickly skim through using it.

The Swagger Editor is an online tool for editing the OAS. You can access it at <https://editor.swagger.io/>. When you first open that in your browser, it will load the example Petstore API. If you have used the editor before, it might not be there, but you can easily load it by going to the **Edit** menu and choosing the **Load Petstore OAS 3.0** option.



On the right-hand side, the editor shows a preview of the documentation that can be produced from the specification. You can play around with the actual specification in the left-hand panel and immediately see the result on the right. Try it out yourself and see. Look through the specification and under the put action for the /pet endpoint in the paths section, change the summary to say **Modify** an existing pet instead of **Update an existing pet**. Note that when you do that, it immediately updates the text in the PUT action of the pet endpoint, as shown in the following screenshot:

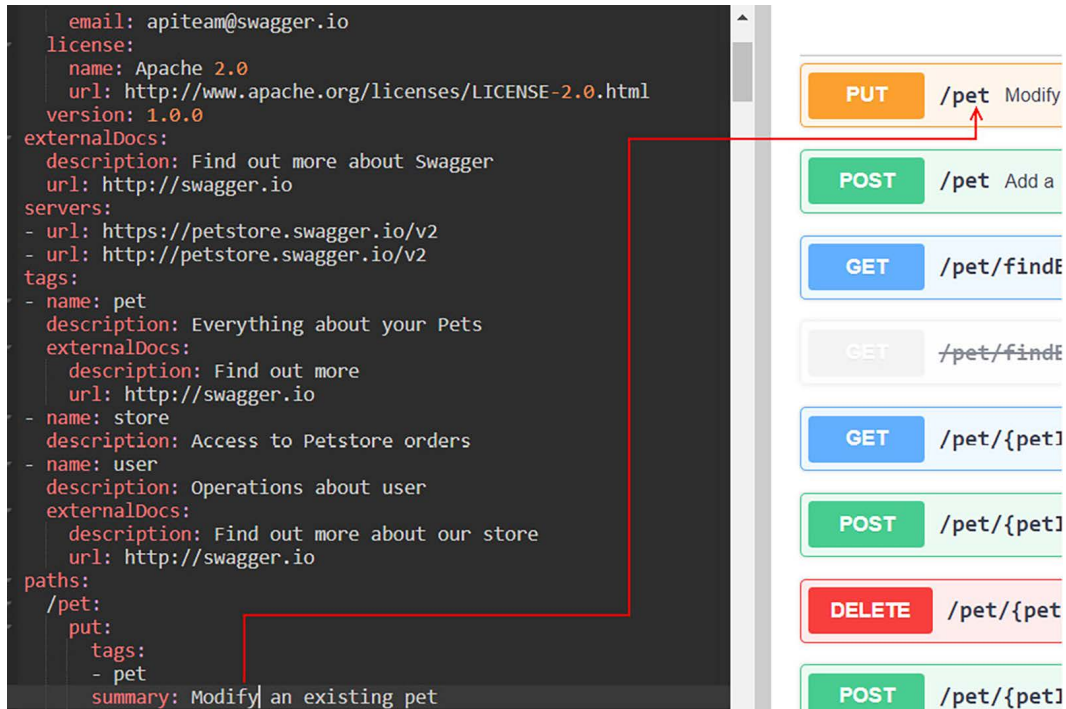


Figure 3.1: Modifying the specification modifies the documentation

Now that you know how to use the Swagger Editor, let's take a closer look at the specification itself to make sure you understand how to create one for your own APIs.

## Parts of an OAS

The first section I want to dig into is the info section. This is where the base data for the API object is defined. At a minimum, you will need to include the title, description, and version options. There are a few other optional high-level tags that help specify metadata about the API. These tags are helpful but not necessary for describing your API. The following screenshot shows different options in the info section:

```

info:
  title: Swagger Petstore - OpenAPI 3.0
  description: |-
    This is a sample Pet Store Server based on the OpenAPI 3.0 specification. You can
    find out more about
    Swagger at [http://swagger.io](http://swagger.io). In the third iteration of the pet
    store, we've switched to the design first approach!
    You can now help us improve the API whether it's by making changes to the definition
    itself or to the code.
    That way, with time, we can improve the API in general, and expose some of the new
    features in OAS3.

    Some useful links:
    - [The Pet Store repository](https://github.com/swagger-api/swagger-petstore)
    - [The source API definition for the Pet Store](https://github.com/swagger-api/swagger-
      petstore/blob/master/src/main/resources/openapi.yaml)
  termsOfService: http://swagger.io/terms/
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
  version: 1.0.17

```

Figure 3.2: Options in the info section of an OAS

The next section in this file is the paths section. This section is critically important and there are a lot of options here. Obviously, it is the place where you define your endpoints, and under each endpoint, you can define the various actions available on that endpoint. You can also define a lot of detail about each endpoint/action combination: things such as what kind of responses this endpoint can give and what kind of data it requires, as well as descriptions of what the API does. As you can see, this section contains the bulk of the API description. This is a pretty simple API and it takes up nearly 600 lines in this specification. This might feel like overkill, but detailed documentation like this helps both developers and QA teams design good tests and create a stable and understandable API. You may not need to leverage every option in here, but it is good to include a lot of examples and details in your documentation, so let's walk through the /pet endpoint to get a bit more of a feel for how to use this tool.

The first thing that is defined is the actual endpoint itself (/pet). The next thing you specify is what actions are available for that endpoint. In this case, it can be called with either put or post. The next things under those verbs are parts of the spec that define some information about the endpoint. These include things such as summary, operationID, and the request body description. Most of these kinds of keys are optional but can be helpful for documenting and organizing your API.

The `requestBody` section defines what needs to go into the body of the API call. In this section, we need to define the content type (it can be either `application/json` or `application/xml` in this example), and then under it, you need to define the schema or structure that this content needs to adhere to. You will notice that the schemas are referenced using a `$ref` key. This is merely a key that points to values that are stored in another part of the document. You could spell out the entire schema directly in this part of the document. However, in this case, the same schema is being used in multiple requests and so it has been moved to another section of the document called `components`. This makes it possible to reference the same thing from multiple places. We will talk about this more soon.

Another important part of any API call is the **response**. The specification defines what should happen for each response that the API can give in the `responses` section. These responses are described under the different HTTP error codes that you want your API to support. The specification also lays out the security options for each action on the endpoint. The actual definition of the security options for the API can be found in the `securitySchemes` section near the bottom of the file.

You can find a full list of all the parameters that you can use in your specifications on the documentation page here: <https://swagger.io/specification/>. In the following screenshot, you will see some examples of a paths specification:

```

paths:
  /pet:
    put:
      tags:
        - pet
      summary: Update an existing pet
      description: Update an existing pet by Id
      operationId: updatePet
      requestBody:
        description: Update an existent pet in the store
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Pet'
          application/xml:
            schema:
              $ref: '#/components/schemas/Pet'
          application/x-www-form-urlencoded:
            schema:
              $ref: '#/components/schemas/Pet'
        required: true
      responses:
        '200':
          description: Successful operation
          content:
            application/xml:
              schema:
                $ref: '#/components/schemas/Pet'
            application/json:
              schema:
                $ref: '#/components/schemas/Pet'
        '400':
          description: Invalid ID supplied
        '404':
          description: Pet not found
        '405':
          description: Validation exception
      security:
        - petstore_auth:
            - write:pets
            - read:pets
    post:
      tags:

```

Figure 3.3: Examples of a paths specification

The paths sections lay out most of the details of what each endpoint can do, but as I noted earlier, the schema for the bodies of these requests is defined in another part of the document called schemas. Let's look at that section of the specification to see how the schemas are set up for the various endpoints in the Petstore API.

## Petstore OAS schemas

On the right-hand panel of the editor, the specification is rendered for you. If you scroll down to the bottom of this panel, you can see a section for the schemas. Find the **Pet** schema in there and click on the **Jump to definition** icon.



Figure 3.4: Jump to schema definition

This will take you directly to the schema definition for **Pet** in the specification. In many ways, the schemas are the core of the specification. They define what the incoming and outgoing data in the API calls need to look like. You can see in the definition that any request that uses the **Pet** schema is required to have a **name** field and a **photoUrls** field. This means that if you send a request to `/pet` url, that response must always include those two fields. A little bit later, we will look at how to use a schema in Postman to automatically check that responses are following the schema rules. In addition, we can see in the schema that the **name** field needs to be of the **string** type, and the **photoUrls** field needs to be an **array of strings**. There are also some other optional fields that you can include, such as **status** and **tags**.

Another interesting thing to note is that the optional **category** field has a **\$ref** that points to another schema. You can see the schema rules for the **category** field by clicking on the **Jump to definition** icon for the **category** schema. If you do that, you can see that the **category** is required to be an object with two properties on it: an **ID** and a **name**.

## Creating your own OAS

Now that you have seen what an OAS file is like, let's look at what it takes to create your own. Creating your own file from scratch will give you a better idea of how the specification works. The example below will just skim over what it takes to create an OpenAPI specification. If you want more details on this, you can check out the documentation here: <https://swagger.io/docs/specification/about/>. OpenAPI specifications can be written in either JSON or YAML format. It is common to use YAML so that is what I will use in the coming example.

In this section, I will walk you through creating an API specification for an imaginary budgeting application. This application will allow users to add new line items to a budget. It will also allow users to update entries once they have been created.

They can also delete an item that they no longer want to track. Our job in this section will be to create an API specification that lays out exactly what this API is going to look like. We will then look at how we can use this specification in Postman.

The approach we will use is known as the **design-first** methodology. You will first design the API specification and then, once that is laid out, we will look at how to use it to drive API development.

## Starting the file

You can write down the specification in the online Swagger Editor or in your favorite text editor. The first thing to create is the general metadata about the API. This is stored in the `info` tag:

```
openapi: 3.0.1
info:
  title: Budgeting API
  description: Manages budget line items
  version: '1.0'
servers:
  - url: http://localhost:5000/budgeting/api
```

Save this information in a file called `budgeting.yaml`.

Endpoint definition is one of the key parts of designing an API. In this case, we know that the API will need to support managing the line items that go into a budget. We will assume that the budget already exists and this API is just used to manage items that are a part of that budget. A logical thing to do, then, would be to have an endpoint called `/items` that will give us information about all the line items in the budget.

However, just getting a full list of these isn't going to be enough. We are also going to need to be able to get and modify information about individual items. In order to do this, we will need another endpoint called `/items/{itemId}`, where `itemId` is a number representing the item that we are looking at.

If you think about the different actions that this API supports, we can then start to build out the methods for each of these endpoints. We are going to need to be able to `GET` the full list of items and we will also want to be able to `GET` information about individual items. We also know that we need to be able to create new line items, and the best way to do that is by doing a `POST` on the `/items` endpoint. In addition, we need to be able to modify and delete individual items, so we will need `PUT` and `DELETE` calls on the `/items/{itemId}` endpoint. You can put a skeleton in place to summarize all this information.

All of the endpoint definitions will go in a section called `paths`:

```
paths:
  /items:
    get:
    post:
  /items/{itemId}:
    get:
    put:
    delete:
```

Now that you have this skeleton in place, you can start to fill out the details of each of these requests. There are several different fields that you can add to each request. You don't need to add all of them, but I would suggest including the `description` and `responses` fields. The `description` field is just a text description of what the endpoint does, but the `responses` field can contain a lot of information about what you expect the response to look like when that endpoint is called.

For example, if you fill out the GET request for the `/items/{itemId}` endpoint, you might make something that looks like this:

```
/items/{itemId}:
  get:
    description: Gets information about the specified line item
    responses:
      '200':
        description: Successful Operation
        content:
          application/json:
```

This response is defined as having a response code of `200`. It also defines the content type as `application/json`. This means that an acceptable response from a call to this endpoint is one that has a code of `200` and one in which the data is in JSON format. This is a start, but it doesn't say anything at all about what that data should look like, so it isn't really that helpful in helping you design the endpoint.

In order to do that, you are going to need to define a schema for what you expect these requests to respond with when they are called. Wrapping your mind around what a schema is and how to use it can be a bit intimidating, but like anything else, it is something that you can learn, so let's dig into it and see what it is all about.

## Understanding the API schema

We can define rules that dictate what the response should look like by defining the **schema**. At its most basic, a schema is just a set of rules that you can apply to the data of the response. These rules include things like what fields are supposed to be there and what kinds of data are allowed to be in each of those fields. A schema is a very important design tool. When we take the time to think carefully about what rules should apply to the data we are sending and receiving, we are working on a design problem. By using a schema to lay out the specific rules for what things an API is allowed to do, we are engaging in a design process.

In real-life application development, you would want to talk to those designing the user interface to see what information they would need. You would discuss what data is needed, what data might be optional, and what data you want to make sure you never get. All these things can be defined with schema rules. Since we aren't actually developing a user interface for this API, we can just make some reasonable assumptions about what this data should look like.

For example, we can assume that each line item will need a transaction date associated with it and will need to specify the amount of the transaction. It will also need a way to specify what category the line item belongs to. We could get more complex and track more information, but let's keep it simple for the first pass at this API and just define these three properties. So, now that we know what properties we want each item to have, how do we set up a schema that defines this?

You can define a schema entry under `application/json`. At this point, you could create the schema directly under that object. However, since most of the responses that we define in this application are going to have similar data in their responses, it would make much more sense to have this schema defined in a place where we can reference it from multiple endpoints. You can do this in OpenAPI by creating a new section in the file called `components` and then specifying a `schemas` entry where you will create each type of schema that you need. If you put together everything we have discussed, you should have a schema definition that looks like this:

```
components:
  schemas:
    item:
      type: object
      required:
        - transaction_date
        - amount
        - category
      properties:
```



```
transaction_date:
  type: string
  format: date
amount:
  type: number
category:
  type: string
```

Note the required entry in there. By default, all properties are considered to be optional, so in this case, we have indicated that we will only consider a response to be valid if it includes the `transaction_date`, `amount`, and `category` properties.

Also note that the OpenAPI specification has a limited number of datatypes, which can then be further restricted to tighten up what is allowed to be there. So, in this case, we specified the type of `transaction_date` as a string but then added the `format` modifier to indicate that this string should follow a date-style format. You could also add modifiers like **minimum** or **maximum** to the amount if you wanted.

Now that you have defined this schema, you need to reference it from the endpoint. You can do this using `$ref` in the endpoint definition. The content portion of the endpoint definition for the `items/{itemId}` endpoint should now look like this:

```
content:
  application/json:
    schema:
      $ref: '#/components/schemas/item'
```

This completes a very basic setup for the `items/{itemId}` endpoint. You will need to follow a similar process to define the other endpoints. The PUT and POST calls will reference the same schema, but the DELETE call will be a bit different. This call does not return the object that is deleted. Instead, it returns an empty object, so rather than having the schema use `$ref` to point to the `items` schema, you can directly set the schema to `type: object`.

The final endpoint we need to consider is the `/items` endpoint. It returns a list of line items and each of those objects can use the `items` schema. However, they will all be wrapped in a list so you can't just directly reference the task schema. Instead, you can define a new schema called `items` (note the plural instead of singular). The type of this item will be array since we expect it to be an array of items. You can specify the items as a reference to the task schema since each item in the array should have that schema.

In the `.yaml` file, it would look like this:

```
items:
  type: array
  items:
    $ref: '#/components/schemas/item'
```

With that, you have defined the schema for these API responses. We aren't quite ready to import this specification file into Postman, but when you do, you will be able to use these schema rules to validate that you are getting the correct responses from the API calls that you send.

## Defining parameters

In the `/items/{itemId}` endpoint, you have specified a path parameter (`itemId`) but have not yet defined it. The OAS requires that parameters like this are defined for each request on this endpoint. You can do this by adding the `parameters` key and then specifying the schema for that parameter. Since this parameter needs to be defined for multiple call types, you can create a new entry for it in the `schemas` section and define the schema there:

```
itemId:
  type: integer
  minimum: 1
```

This specifies that `itemId` must be an integer and that it cannot be less than one. You can then reference this schema from the parameter along with specifying the name of the parameter and the fact that it is a required path parameter. Pulling that all together, the code for it would look like this:

```
parameters:
- in: path
  schema:
    $ref: '#/components/schemas/itemId'
  name: itemId
  description: Id of the line item
  required: true
```

This specifies the details for this parameter and is placed under the GET, PUT, and DELETE definitions for the endpoint. Note that the `- in` parameter has the `-` character in front of it. This is because parameters are considered to be an array object and, in YAML, arrays are specified with a dash.

## Describing request bodies

When sending a POST or PUT request, you need to specify some data in the body of the request. In a similar way to defining the responses, you can define the content of a request body for these requests. The request body for the POST response in the to-do list API might look like this:

```
requestBody:
  required: true
  content:
    application/json:
      schema:
        $ref: '#/components/schemas/item'
```

The `required: true` item means that you must specify a body when sending this request in order for it to be a valid request. The rest of the definition is the same as that for a response. You specify the content type that is allowed (and you can specify multiple content types if the API supports more than one), and then you specify the schema that dictates what rules the body needs to follow. In this case, you can use the same item schema that was defined earlier since that same data format is what you will need to specify when creating a new task.

The PUT request body definition is the same as the one we just defined for the POST request. You can copy the definition that you just made into the PUT request definition.

We have now covered most of the parts of the OAS, but one other commonly used item is `examples`. They are not required for an API definition, but they are nice to have, so let's look at how to use them.

## Using examples

So far, we have defined the behavior and schema of the API, but it is also nice to give some examples so that API users can understand a bit better how the API works. Examples are just illustrations of possible ways to use the API. They are not rules that must be followed, but they can be really helpful to have in place. They can help human users understand how to use the API, and they can also be used as a basis for creating automated tests.

Examples can be added to a `requestBody` or to a response. Let's add one to the `requestBody` that we defined in the previous section. You can do that by adding an `example` key to the `application/json` content type item and then specifying examples of the properties that need to be sent. The code for that would look like this:

```
content:
  application/json:
```

```
schema:
  $ref: '#/components/schemas/item'
example:
  transaction_date: '2024-06-01'
  amount: 123.12
  category: 'groceries'
```

This example shows you one way to specify the body of the request. In this case, we only added one example, but what if we wanted more—say, for some different categories or dates? You can do that by changing the key to `examples` (with an `s`) and then giving each example a name and value, which would look like this:

```
examples:
  Groceries:
    value:
      transaction_date: '2024-06-01'
      amount: 123.45
      category: 'groceries'
  Rent:
    value:
      transaction_date: '2024-06-01'
      amount: 1500
      category: 'rent'
```

Examples can also be defined for responses as well as in a few other places in the definition. They are all defined in a similar way, so you should be able to figure out how to do them.

With that, we have covered most of the main features of the OAS. There are, of course, additional details that I haven't been able to show you here, but by using what you've learned in this chapter, you should be able to create your own API definition file. Creating a file like this is helpful for thinking through the API design and for documenting what your API should do, but it can also be used for a lot more than that. In the rest of this chapter, we'll look at how you can use this kind of file in Postman to help you with your testing work.

## Using API specifications in Postman

You can use API specifications to simplify work in many areas. In this section, I will show you some things that you can do with them. You will learn how to create mocks and tests from a specification with only a few clicks.

I will also show you how to use them to do some validation so that everything in your request and elements are consistent.

If you have been following along, you should be able to import the file you just created, but if not, you can download the file from the GitHub repository for this course (<https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/master/Chapter03>). Using an OAS in Postman is as easy as importing it and following a few setup steps, as follows:

1. In Postman, choose the **Import** button at the top left of the application.
2. In the file browser, navigate to where you saved the `budgeting.yaml` specification file and click **Open**.
3. On the resulting dialog, select the **OpenAPI with a Postman Collection** option and then click on **View Import Settings**.
4. Ensure that the **Parameter generation** option is set to **Schema** and then go back to the **Import** dialog.
5. Click on **Import**. This will automatically import the API and create a collection.
6. Once the import has completed, go to the **API** section in the navigation tree, expand **Budgeting API** and then **Definition**, and click on `budgeting.yaml` to see the raw definition file that you just imported.

You should see that Postman has created a collection for you with all the endpoints from the specification file. We started with a design-first approach to creating this API, so at this point, we don't have an actual API to make calls to. However, we can create a mock server in Postman. A mock server is just a server that returns some hardcoded responses to our requests. We will set one up now so that you can see how to generate them from a schema, but don't worry too much about how exactly they work and how to use them. We will return to mock servers in a future chapter and dive into a lot more details on how to set them up in the backend and how to use them for effective testing, so don't worry too much about understanding the details of how they work right now.

## Creating a mock server

Mock servers in Postman are based on examples in collections. During an import, Postman should **create** a collection on the API tab and another one on the **collection** tab. Go to the one on the **collection** tab and, on the ellipsis menu for that collection, select the **Mock collection** option:

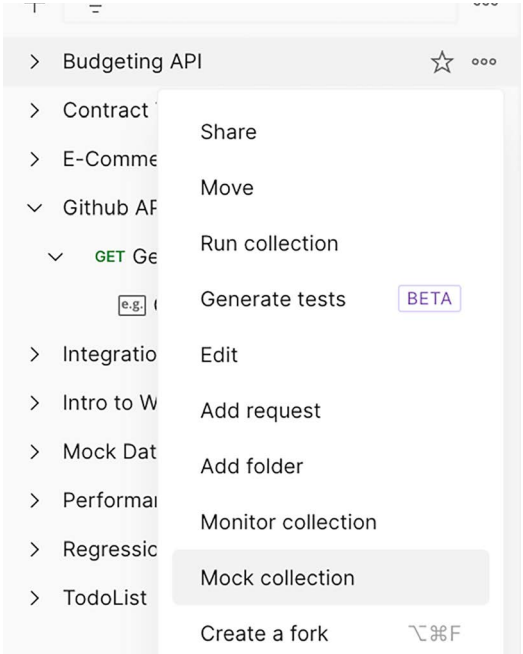


Figure 3.5: Mock collection

Give the mock server a name (something like Mock Budgeting API) and click on **Create Mock Server**. That’s all it takes to create a mock server, so now let’s look at how to use it:

- 1. Copy the URL that Postman shows.
- 2. Go to the **API** section of the **navigation** panel and, under the **Budgeting API**, select the **Budgeting API** collection.
- 3. Go to the **Variables** tab of this collection.
- 4. Paste the mock server URL into the **Current Value** field of the `baseUrl` variable.
- 5. Add the base URL ending to match the initial value. For example, in the screenshot below, you can see that the initial value ends with `budgeting/api` and so does the current value:

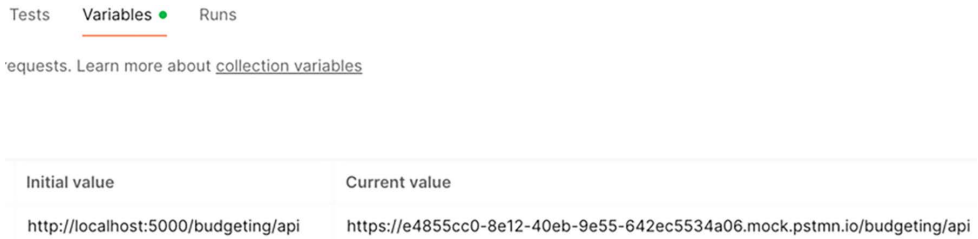


Figure 3.6: Using the mock server URL

Make sure to save the changes. You are now ready to take a look at how we can use the OpenAPI specification to help us validate our results.

## Validating requests

API specifications can be used to help you validate requests. With request validation, Postman checks that all the data sent and received matches the API specification. It might be hard to visualize what I mean by that sentence, so let's look at it in action:

1. In the **Budgeting API** collection, under the API definition, expand the `item` folder and then the `{itemId}` folder and click on the request that gets the information for the specified line item.
2. Click **send**.

You will get back a response that uses the example. The response will look something like this:

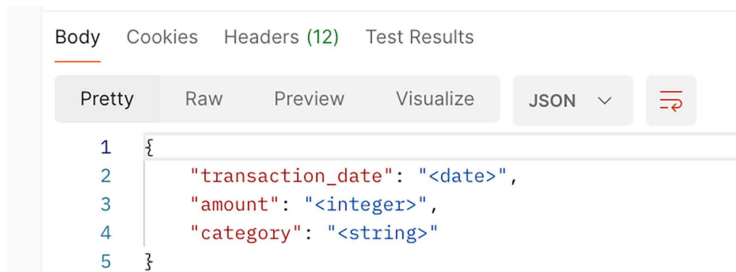


Figure 3.7: Mock server response

Although you can see that the example values are generic, you can also see the power of using schemas to help automate some of your work. With just a couple of clicks, you were able to set up a mock server and get back some sample results for these requests. The ability to do this kind of automation is one of the reasons that using schemas like these is so helpful.

## Summary

API specifications give you a lot of power in your testing. You can use them to keep documentation, tests, and even underlying code all in sync with each other. In this chapter, you learned how to use them to reduce the work in generating mocks and collections in Postman. You can also import API specifications that others may have made and use those in Postman. You have learned how to leverage an API specification to help you quickly create tests and mocks in Postman. Taken as a whole, you have gained a lot of useful skills for using an API specification to help with testing and designing high-quality APIs.

Creating and maintaining API specifications can be a bit of work, but they help with many aspects of API building. You are now able to create and use OpenAPI specifications in Postman!

The ability to use API specifications will let you create APIs that are characterized by consistency. This is an important factor in API design and quality, but we also want to be able to automate things. In the next chapter, I will take you through some of the theory of how to create good test automation. API specifications help give some structure and organization to API testing but there is so much more that you can do with Postman. Let's dive into that in the next chapter!

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.com/invite/nEN6EBYPq9>







# 4

## Considerations for Good API Test Automation

In 1811, an angry group broke into a factory in the town of Nottingham in the UK. They started smashing machinery and equipment and destroyed much of the factory before fleeing. These men had been angry and upset for some time. They had issued manifestos and sent threatening letters, but on this day, they took the even more radical step of destroying the machines in this factory. Why were they so upset?

Well, to sum it up in a word: automation.

These men were skilled artisans who had dedicated their lives to the craft of making clothes, and they did not like the new machines that were able to make clothes much quicker and cheaper than they ever could. They claimed to be following the orders of a man named Ned Ludd and called themselves Luddites. This term has entered our modern vocabulary as a description of those who dislike new technology, but the origin of it goes back to the protest movement in which artisans were afraid of losing their jobs to automation.

I rather enjoy the philosophy of automation – when is it helpful and when is it not? But that discussion is outside the scope of this book. What I want to focus on in this chapter is how we can best use test automation. The Luddites framed the debate in terms of a conflict between artisans and automation. I want to reframe that debate a bit. I want to help you become an artisan *of* automation. Automation is here to stay, and I think that, overall, it is a good thing. It allows us to extend our capabilities as humans and to do more good than we could have otherwise, but that doesn't mean that all automation is good.

Automating something isn't just a matter of taking a manual task and turning it into a set of algorithmic steps that a computer can do. It requires careful consideration to ensure that it is actually helpful. Doing the wrong thing faster isn't helpful; it's destructive. Creating automation that takes more work to maintain than the work it saves isn't doing any good either. There are plenty of ways to create bad automation. In this chapter, I want to help you understand how to create good automation.

This chapter will lay some foundations that will be built on a lot in future chapters, but you will still find that you have a lot of practical skills to use by the time you get through the material here. We will be covering the following topics in this chapter:

- Exploratory and automated testing
- Organizing and structuring tests
- Creating maintainable tests

By the end of this chapter, you will be able to create useful test automation, choose what to include (or not) in automated API tests, create well-structured test suites, use variables, understand variable scopes in Postman, use best practices when deciding where to create Postman variables, and explain the importance of logging and reporting for creating maintainable tests.

Let's get started!

## Technical requirements

The code that will be used in this chapter can be found at <https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/master/Chapter04>.

## Exploratory and automated testing

It is easy to think of test automation as a way to quickly do the things that we could do in a manual way. There can be times when this is true. If you are doing some tedious, repetitive work that requires doing the same thing over and over again, by all means, automate it if you can! However, there is a danger that comes with this kind of thinking as well. If you start to think of automation as replacing what humans do, you will end up making poor automation. Manual and automated testing may have the same goal of reducing the risk of shipping bugs to clients, but the way that they achieve this goal is radically different.

For example, a manual test that is meant to help you figure out if a change to the API has caused issues will be guided by what you, as the tester, know about the change that has been made.

It will also be influenced by the testers themselves. A tester who is familiar with an API will observe things that someone without familiarity with the API will not be able to notice. You will notice things and possibly even slightly alter what you are doing. Even if you are trying to follow a strict script (which is not a form of manual testing I would generally encourage), there will be variations and changes in what you, as the tester, do from one time to the next. There might be minor changes in the system that you know are fine and you just skip over them. There might be other things that you see that aren't related to the change at all that you make note of. Even when scripted, testing being performed by humans in this way always has an element of variation built into it.

On the other hand, an automated test that is meant to help you figure out if a change to the API has caused issues will do exactly the same things it did last time and check all the same things in the same way. It will not notice any incidental changes that might be problematic. If there are minor changes that don't really matter, it won't know that and might fail for the wrong reason. Compared to exploratory testing, automated tests are less comprehensive and more brittle. But it's not all bad news. There is a reason we value automated tests as an industry. They can be much faster to run and can look at variations and details that can be hard to see when you're doing exploratory testing.

I have summarized some of the differences between human-based testing and automated (or machine) based testing in the following table:

	Human-Based Testing	Machine-Based Testing
Benefits	Better judgement More dynamic Humans notice things outside the scope of the test	Faster Extends human capabilities
Cons	Slower Can be tedious	Can be brittle and hard to maintain Only notices things explicitly specified

Table 4.1: Exploratory/automated testing comparison

A good test strategy will include a good mix of automated and human-based tests. We should be careful not to pit automated testing against human-based testing. Exploratory testing with a high degree of human involvement can still use various automated tools, and automated testing always requires some degree of human involvement.

What I want to emphasize here is that there are certain things that are important to consider when creating automated tests. Since they can be brittle and hard to maintain, it is important to structure them well and to think about how they will be maintained. By understanding some of the tradeoffs between testing with higher and lower amounts of human involvement, we can better focus on how to design useful test automation.

## Exercise – considerations for good API test automation

In the next section, I will go over what I think are some approaches that you can use to create good test automation – but don't just take my word for it. Before you read through the next section, pause for a moment and think about what things you think are important for good test automation.

Write down a list of some factors (maybe two or three) that you think would be important when creating test automation. After reading through the next section, compare your list to the things I talk about in that section. What is the same? What is different? What do you think of the differences? I will be sharing a few considerations here, but by no means are they the only possible things to think about. Entire books have been written on test automation. The quick review I will provide here certainly does not cover everything.

## Writing good automation

Test automation can happen at many different places in an application. You can have automated unit tests, automated UI tests, and, of course, automated API tests. There are some general principles that apply to all test automation, but the application of them will differ depending on the type of automated testing you are doing and even the tools that you are using to do the automation. Let's start with some of the general principles for writing good test automation and then we'll look at how to apply these in the context of API testing and then at how to implement them in Postman.

We already learned that automated testing can be brittle, so what things can we do to help mitigate that? The first principle for writing good test automation is remembering that **software changes**. It seems obvious to say, but I have seen tests that are written in ways that make it very hard to change them later. If we know that software changes, we should write our tests in a way that expects that and makes it easy to evolve them as the software they are testing evolves.

The next principle for writing good test automation is that **tests should be repeatable**. Sometimes when a test fails, the reason for it is immediately obvious, but often we need to do some debugging to figure out the reason for the failure. This means that we need to be able to easily rerun our tests multiple times.

In general, tests should not be highly coupled with other tests, and it should be easy to rerun a test in isolation from all the other tests.

Related to this is the principle that **tests should be easy to debug**. Tests should print out relevant information when they fail. Tests that never fail aren't actually the most useful tests. The most useful tests are the ones that fail when the code breaks and that make it easy to figure out where the failure is coming from. An example of where you need to use this principle is when using random inputs. You should always provide some way to get insight into what random value the test was populated with.

Another important principle to remember is that **tests should be well organized**. It is much easier to figure out what you have and haven't tested in well-organized test suites. It is also easier to target running just a subset of tests. Another benefit is that it is easier to create helpful reports. Automated tests can be run over and over again. Automation can run much more quickly than a manual test, and computers don't get bored of doing repetitive tasks. You want to be able to leverage this power in your tests. In order to do this, you will need to have some kind of reporting in place that lets you know if there are test failures. You want these reports to be actionable so that they can help you pinpoint where there might be bugs, or where you might need to fix tests. This is much easier to do if your tests are well structured.

Something else to keep in mind when creating automated tests is that **it is okay to be repetitive**. In software development, there is the principle of **Don't Repeat Yourself (DRY)**, but that is less of a concern in automated test development. It is often helpful to have test inputs close to the test so that you can more easily understand what a test does and how it might have failed. Don't always jump to putting repeated code into common methods. Sometimes it is okay to be repetitive in automated testing.

I have just gone through some of the principles that can help you create good test automation. In a moment, we will look at how to apply those principles to API testing, but first, let's look at some of the different types of API tests that you can create.

## Types of API tests

There are several types of API tests that lend themselves well to automation. **Contract testing**, which we will dive more deeply into in *Chapter 13, Using Contract Testing to Verify an API*, is a powerful form of API testing that you can use to verify that your API is meeting a given contract, which is usually specified in some sort of specification language such as OpenAPI.

You can also create **integration** or **workflow**-style API tests where you are testing that you can accomplish a certain workflow. This kind of testing would typically involve calling multiple endpoints. For example, you might **POST** something to create a new item and then call another endpoint to verify that that item has been correctly created and is also accessible in other parts of the system. Another example would be to **GET** an object such as a shopping cart and then use information in that response to **GET** some related information, such as details about the products in the cart.

Of course, you can also just create simple **endpoint tests** that call an endpoint with various kinds of inputs and verify the results. One important thing to consider here isn't so much a type of API testing as it is a style of test that should be run. You should check some negative test scenarios. What happens if you put in bad values or incorrect data? Like any good testing, API tests should not just consider what happens when things go right. They should also look at what happens when things go wrong.

One main category of tests where things can *go wrong* is **security testing**. I will talk a bit more about security testing in *Chapter 5, Understanding Authorization Options*, when I get into authorization and authentication, but it is an important style of testing to consider in APIs. APIs expose a lot of power and flexibility to end users (and that is partly why many companies make them), but in doing so, it is easy to miss things that could allow users access to things they should not have access to. Many security breaches in recent history have come from problems in APIs. I can't dive too deep into that topic in this book, but I will show you some basic things that you can do in Postman to check for API security.

Another type of testing that is worth mentioning here is **performance testing**. Postman isn't really designed as a performance testing tool, but you can get some basic performance data out of it. I will not be talking much about performance in this book, but it is an idea worth keeping in mind as you consider API quality. Performance can have a big impact on the customer's perception of quality, and APIs can often be performance bottlenecks as they involve network latency and other difficult-to-control factors.

As you can see, there are a lot of different ways to approach testing an API. Let's look at how you can use some of the principles for creating good test automation when creating different types of tests in Postman. We will start with how you can structure and organize tests in Postman.

## Organizing and structuring tests

As the saying goes, an ounce of prevention is worth a pound of cure. This is good general life advice and is also true in test automation. Taking a bit of time to structure and organize your tests when you are starting will save you a lot of time later when you are trying to understand test failures or reports on test effectiveness. Postman understands this philosophy and makes it easy to keep tests well organized.

It is too easy to spout off a bunch of theory that you will skim over and not fully understand. In order to keep this practical, I will try to walk through a concrete example. I will once again use the Star Wars API for this (<https://swapi.dev/>). So, how would you go about structuring the tests for this API?

### Creating the test structure

In Postman, collections are used to organize the tests. One way you can think of a collection is as a folder that you can use to collect items, such as other folders and tests. You may already have a Star Wars API collection if you did the case study in *Chapter 1, API Terminology and Types*, but if not, I have provided a collection in the GitHub repository for this chapter. You can import that collection into Postman by following these steps:

1. Go to this book's GitHub repository (<https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition>), click on the **Code** drop-down arrow, and download or clone the code.
2. Go to the **File** menu in the Postman application and choose the **Import** option.
3. In the file browser, navigate to where you downloaded the files from GitHub.
4. Open the `Star Wars API_Chapter4_initial.postman_collection.json` file.

This will create a collection for you that you can use for the rest of this section. You can find the collection in the **collections** navigation tree in Postman.

The Star Wars API has six different resources. We could just create requests directly in the newly created collection for each of them, but that would get a bit messy. For example, the `/people` resource gets a list of all the people in the service, but you can also get information about specific people. If we want to be able to test different things, we will need to include several different calls to the `/people` resource.



Instead of creating a request for this resource, I will create a folder in the Star Wars API collection for the /people resource. I can do this by clicking on the **View more actions** menu beside the collection and choosing the **Add folder** option from that menu. I will then name that folder People. I can repeat this for the rest of the resources in this API (films, starships, vehicles, species, and planets):

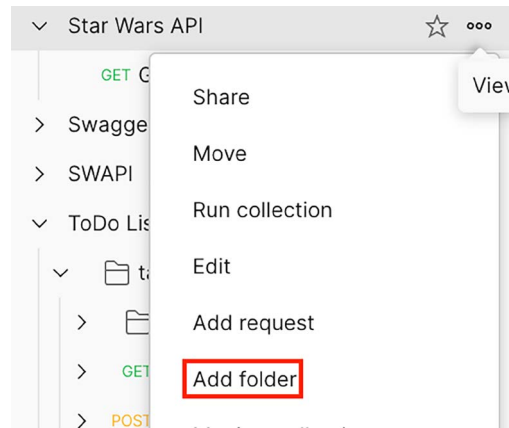


Figure 4.1: Adding a folder to a collection

Now, with folders in place for each endpoint, I want to stop and think about what type of tests we are creating here. Although this service does provide a schema for the different resources that I could use to verify some of the results, it doesn't provide the kind of specification that I would need to create contract tests, so I won't worry about that kind of testing for now. I also won't worry about performance and security testing on this API. That leaves integration testing and endpoint testing. We will want to verify the various endpoints and so will want endpoint tests. There are no direct workflows that we need to check here from a business sense since this is just a "for fun" application, but there are still some interesting workflows that we might want to consider. For example, each of the resources cross-links to other resources. A call to a /planets endpoint gives links to people who live on that planet, along with other links. Each resource does this and provides links that point to other resources. This kind of linking creates some interesting possibilities for testing. You could check a planet's resources and see that they point to certain people, and you can check those people's links to ensure that they have references to that planet. Tests like this could be a simple way to validate data integrity.

It seems like we want to create both kinds of tests, so the next thing I will do is create a folder in the collection called **Integration Tests**. Taken as a whole, these folders provide the basic structure for how we want to approach testing this API, but how do we organize these tests?

## Organizing the tests

The first thing you need to do is to put the endpoint tests for each resource in the appropriate folder. I will start with the `People` folder. Click on the menu beside the folder and choose the **Add request** option:

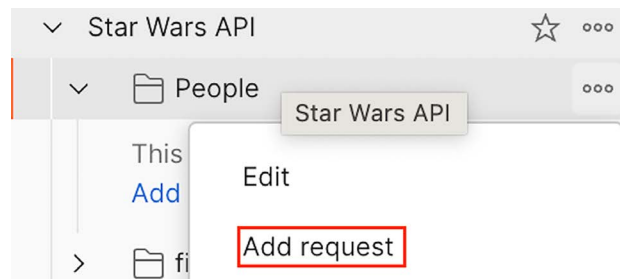


Figure 4.2: Adding a request to a folder

Name the request `Get All People`, click on the request in the tree, and set the request URL to `https://swapi.dev/api/people/`. This endpoint gives back the list of all people. In *Chapter 6, Creating Test Validation Scripts*, I will show you how to create checks to validate this data, but for now, I'll just focus on how to organize these tests.

Each person in this list has their own endpoint, and we could create a request for each one. However, doing that could get very unwieldy. Instead, I will just sample one of the individual people endpoints. Once again, add a request to the folder and name this one `Get Person Details`. Set the request URL to `https://swapi.dev/api/people/{{peopleId}}`. Note that, in this case, I have set the ID to be a variable enclosed in double curly braces. This is because I don't want to hardcode a particular value, but rather want to be able to try different characters at different times.

When it comes to organizing tests, sharing variables between them is very important. You have typed in a variable, but when you hover your mouse over it, you will see that it is an unresolved variable:

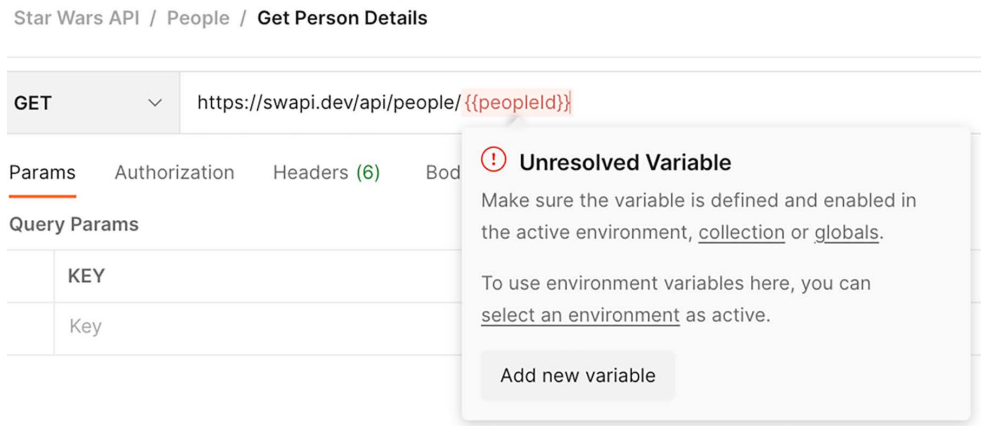


Figure 4.3: Unresolved variable

This warning is here because, by surrounding it with double curly braces, you told Postman that you want to use this `peopleId` variable, but you have not assigned that variable any value. There are a few different places in Postman where you can store variables. The first place is in the collection itself. These are aptly named **collection variables**, and we will discuss them in more detail later. In this case, I want you to save this variable as an environment variable, so let’s look at how we can do that now.

Environments

In order to create an environment variable, you will need an environment in which you can save the variable. You can create a new environment in Postman by following these steps:

- 1. Click on the **Environments** section in the navigation panel.
- 2. Click on the **Create Environment** link.
- 3. Name it **SWAPI Environment**.
- 4. Add a new variable called `peopleId` with an initial value of 1:

SWAPI Environment				
	VARIABLE	TYPE ①	INITIAL VALUE ①	CURRENT VALUE ①
<input checked="" type="checkbox"/>	peopleId	default ▾	1	1

Figure 4.4: Adding a new environment variable

5. Click on **Save** to add this new environment and then close the **Manage Environments** tab.

Now that you have the variable defined in an environment, return to the **Get Person Details** request. The variable will still be unresolved because you need to tell Postman which environment you want to use. To do that, click on the dropdown near the top-right of the screen and choose **SWAPI Environment** from the list:

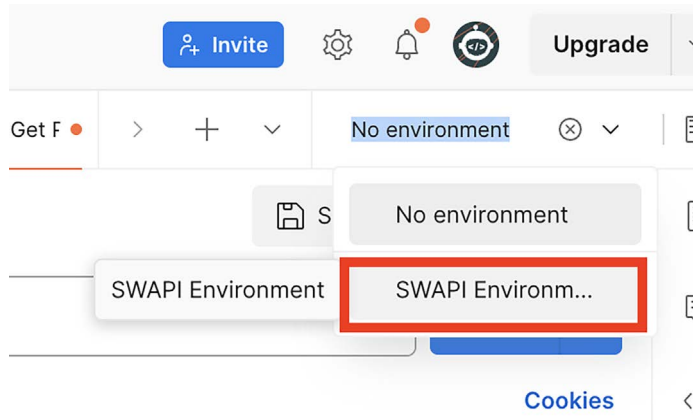


Figure 4.5: Choosing an environment

Now that you have set the environment, if you mouse over the `peopleId` variable, you should see that it is set to 1. Environments are a great way to organize data for your tests. In *Chapter 6, Creating Test Validation Scripts*, we will cover the different ways that we can share data and manipulate variables in the environment, which will make the concept even more powerful. Environments allow you to manage your variables in one place. This is very helpful if you need to make changes or just need to understand what variables you have set up. Instead of looking through every endpoint to see what might be defined and trying to decide if you can change things in them, you can collect all the variables in one place and see what you can change at a glance.

## Collection variables

I mentioned earlier that you can also store variables in the collection, so let's look at how that works. You have created a couple of requests already and each of them starts with the base URL `https://swapi.dev/api`. Instead of repeating that on every request that you put in the collection, let's turn it into a variable:

1. Go to each of the requests that you created earlier and replace the `https://swapi.dev/api` URL with a variable called `{{baseUr1}}`.
2. In the collection navigation pane, click on the collection.

3. Go to the **Variables** tab for the collection and create the `baseUrl` variable, setting its initial value to the base URL (`https://swapi.dev/api`).
4. **Save** the collection.

The variable will now be defined in all the requests that you are using it in. You can also use that variable when defining any new requests for this API.

## Choosing a variable scope

You now know of two different places you can store variables in Postman (environments and collections). There are a couple of other places where you can store variables (for example, you can make global variables), but how do you decide where to store them? How would you decide whether you should put a variable into an environment, in a collection, or somewhere else? Well, in order to answer that question, let's look at the different places where you can have variables in Postman and understand how they relate to each other.

## How scopes work

When you store a variable in a collection or an environment, Postman will give that variable a **scope** that corresponds to where you have stored it. When you're trying to resolve a variable, Postman will look for it in a defined order in each of the scopes. Scopes allow a variable to be used in broader or narrower parts of the application. If you define a variable in the global scope, you can use it in any request anywhere in the application, but if you define it at the narrowest scope, it will only be available during one particular iteration of a test. So, what are the available variable scopes in Postman? In order from the broadest to the narrowest scopes, they are as follows:

- Global
- Collection
- Environment
- Data
- Local

When resolving a variable, Postman will use the narrowest scope in which that variable exists. Let's take a look at a concrete example illustrating this.

You have already created the `baseUrl` variable in the collection scope. As a test, let's create a variable with the same name in SWAPI Environment. You can do that by following these steps:

1. Click on the **Environment Quick Look** icon beside the environment dropdown.
2. Choose the **Edit** option to make changes to the environment:

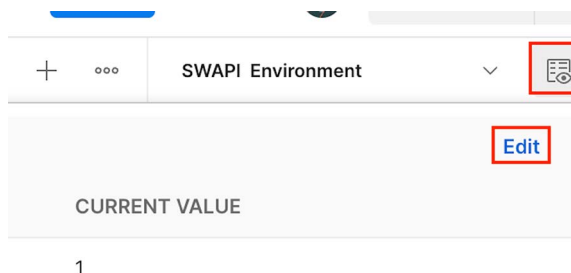


Figure 4.6: Editing an environment

3. Add a new variable called `baseUrl` and give it a silly value such as `bob`.
4. **Save** the environment.
5. Go to one of the requests and mouse over the `baseUrl` variable.

You can see that it now has a value of `bob`, and you can also see that the scope of this variable is **Environment**. Even though, in the collection, the variable with the same name contains the site URL, the value being used comes from the variable defined in the environment since an environment has a narrower scope than a collection:

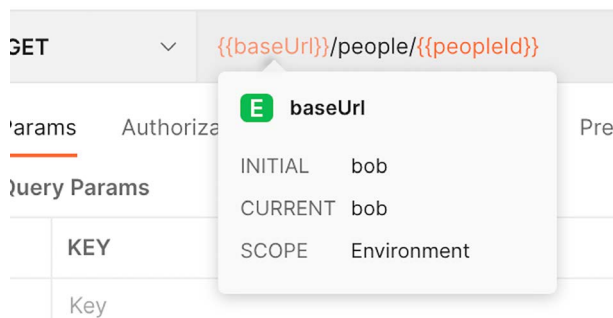


Figure 4.7: Variable values coming from the narrowest scope

Don't forget to go back into the environment and clean things up by deleting the silly variable that you made! To do that, edit the environment again and then click on the **Delete** icon that shows up beside the variable when you mouse over the field:


	Variable	Type	Initial value	Current value	...
<input checked="" type="checkbox"/>	peopleId	d... ▾	1	1	
+ <input checked="" type="checkbox"/>	baseUrl	d... ▾	bob	bob	 ..
	Add new vari...				Delete

Figure 4.8: Deleting a variable

You now understand how Postman determines which value to use when variables are defined in different scopes, but how do you know which scope to create a variable in? Let's do a quick overview of the different scopes and discuss when you would want to use each of them.

Global scope

The first scope we will discuss is the **global** scope, which defines variables that are accessible globally or anywhere in the application.

You may want to use global variables when experimenting with sharing data between different tests or collections, but in general, you want to try and avoid using global variables. They are, by definition, available everywhere, and if you use them, it is likely that you will end up eventually giving a variable in another scope the same name as the global variable. Postman will, of course, use the narrower scope, but this can still lead to confusion, especially when you're trying to debug failures.

You can add and edit variables in the **global** scope by using **Environment Quick Look**, and then choosing the **Edit** option in the **Globals** section:

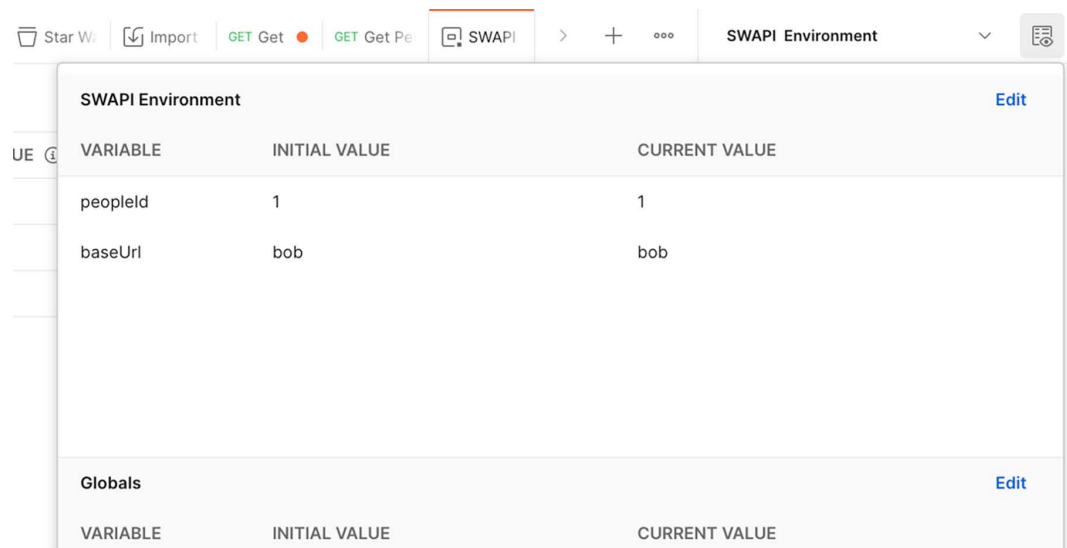


Figure 4.9: Editing global variables

Variables in the global scope are the most widely available and the last ones that Postman uses when trying to resolve a variable value. Let’s continue with the narrower collection scope.

Collection scope

Variables defined in the collection are available for all the requests in that collection. They are not available in other collections, but any request in that collection can use them. You want to create collection variables when you need to share the data between multiple requests in the collection, and that data will stay the same. An example of this is an application URL. Every request will need it, and it should stay consistent. Another example would be doing some setup work with a pre-request script (I will show you how to do this in *Chapter 6, Creating Test Validation Scripts*) and then storing that data in a collection variable.



## Environment scope

Environments allow you to define a set of variables that belong to a certain context. For example, many companies will have testing sites and production sites. You might also have a staging environment or want to run tests on a local build. Each of these environments might require some minor changes to be made to the variables being used, so in these cases, you would want to create variables that live in the different environments.

For example, if you had staging, testing, and local sites that you wanted to test against, you might define an environment in Postman for each of those sites, and then in each of those environments, create a URL variable that points to the correct endpoint for each of those environments. It is important to note that, in these cases, you should try not to also have the same variable defined in the collection. This will lead to confusion and is not a good practice.

Another time you might store variables in an environment is if you want to try some API endpoints as different users. You might want to try as an administrator and a regular user, for example, so you could create environments for each of those users. These will store the login information for them and maybe some specific variables that store expected results for those different users.

If you find yourself with only one environment for a collection, you are probably better off just creating variables directly in the collection instead of adding the additional complexity of an environment.

## Data scope

Most variables will be defined in collections and environments. However, in *Chapter 7, Data-Driven Testing*, I will show you how to do data-driven testing in Postman. This kind of testing imports data into Postman from a CSV or JSON file. Postman will create variables in the data scope from this imported data. You cannot create these kinds of variables yourself within Postman, but you should be aware of the kinds of variables that you have defined in your collections or environments when you are creating your input data files. The scope of data variables is narrower than that of collections or environments, so any variables defined in this scope will overwrite values defined in those broader scopes.

## Local scope

The narrowest scope in which you can define variables in Postman is the local scope. You can only create variables in this scope using request scripts (which I will talk about in *Chapter 6, Creating Test Validation Scripts*).

They are temporary variables that do not persist between sessions, but since they are the narrowest scope, they do allow you to override values that have been set in collections or environments. Sometimes, this is desirable as it lets you override a variable in one request while still using it in all the other requests in a collection, but you should be careful that you are not accidentally overriding them either.

Variables and the different scopes that they can be stored in give you a lot of power, control, and flexibility when it comes to creating and organizing tests. They also help with creating tests that continue to be valuable over time.

## **Exercise – using variables**

Hopefully, at this point, you have a pretty good grasp of the various variable scopes, but I would encourage you to try this out, both to cement these concepts in your mind and to prove to yourself the order in which Postman resolves the scopes. Create a variable in a few different scopes and play around with its values to make sure you understand how Postman is using that variable in requests.

Effective use of variables is an important factor in test automation. Take some time and make sure that you understand this. Good management of this will help you create maintainable tests, but there are other factors that are important as well. Let's look at some of them.

## **Creating maintainable tests**

One of the things that frequently gets forgotten in conversations about test automation is that it takes time and work to maintain. The “sales pitch” for test automation is that we can run the same test over and over again “for free” but, of course, that is not true. Leaving aside the hardware and software costs of running these tests, there are maintenance costs. Tests don't always pass. Sometimes, failures are due to finding a bug, but other times, it is just because the code has changed and the test needs to be updated, or because of some kind of flakiness in the system that we don't need to worry about too much. At the start of this chapter, I mentioned that one of the principles to keep in mind for well-written tests is that they should be easy to debug. They assume that there will be failures in the future that need to be debugged. So, how do you set yourself up to make sure that your tests are maintainable and easy to debug?

## **Using logging**

One of the ways in which you can make it easier to figure out failures is by having good logging options set up in your tests. You can view the logs in Postman using the console.

You can either open a standalone console by going to the **View** menu and choosing the **Show Postman Console** option or you can open an in-app console by using the **Console** icon at the bottom of the application:

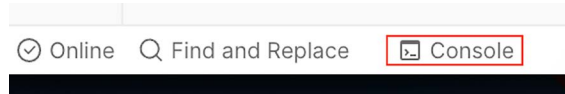


Figure 4.10: Opening the Postman Console

The console will show you further details about the requests, along with any errors or warnings. It is also the place where any `console.log()` commands that you set up in your scripts will print.

Adding debugging information to a test is a bit of an art form. It can be tempting to think that more is better and so you should just log out information at every step of your test. However, that usually leads to information overload where there is so much information reported that it becomes hard to find the relevant information. In general, I would discourage tests that have console logging in them. If you are trying to debug a particular test, it is a great idea to add in some temporary `console.log()` statements to help you figure out what is going on, but you shouldn't leave those in permanently. You should set up variables and data in your test to make it easy to add logging as necessary, but don't keep log statements in.

Maintainable tests also include some things that I have already talked about, such as writing good documentation and organizing your tests in a logical way, but another consideration for writing maintainable tests is having good test reports.

## Test reports

Any automated test system needs to report the results in some way so that you can find out whether there are any issues to investigate. Postman will report on results directly in the application or command line. Many of the reporters built into Postman require that you have an Enterprise or Business plan.

However, there are some free reporting tools available, including a number of third-party reporters. **newman-reporter-htmlextra** (<https://github.com/DannyDainton/newman-reporter-htmlextra>), written by Danny Dainton, is probably the best one available. This reporter works with the Postman command-line runner known as **Newman**, which I will show you how to use in *Chapter 9, Running API Tests in CI with Newman*. In that chapter, I will also show you how to use the test reporter, but for now, keep in mind that good test reporting will help with creating maintainable tests.

## Creating repeatable tests

Another important factor for creating good test automation is having repeatable tests. This means that you can run the test multiple times in a row and get the same result. With API testing, this is usually not a problem when testing GET requests, since they should be idempotent. You do need to be careful if you are sharing variables between tests, but usually, GET requests are naturally set up to be repeatable.

Other types of tests, like those testing POST or DELETE calls, might need a bit of thought to ensure that they are repeatable. For example, let's think about testing a DELETE request. If you are testing with a testing or staging environment, you might already have data in there that you can use for testing purposes. It might be tempting to test a DELETE call by pointing at a resource that you know is already set up on the testing server, but will that be repeatable? Once you have deleted the resource, it is gone from the testing server and, if you try to run the DELETE call again, there will be nothing for it to delete. It would be better to have the test first create the object you want to delete and then test that the delete method works. This way, no matter how many times you run the test, it will always have something to delete.

Another factor to consider in ensuring that tests are repeatable is thinking about what else might be going on while your test is running. If there are other people or test processes using the server while your tests are running, you need to make sure that you are running your tests in a way that is isolated from those changes or that at least allows you to set up an initial state the way that you need.

I often find that in more complex testing situations where there might be multiple users on the server at the same time, it is helpful to have a setup section in the tests. This section can be set up so that it can be run along with any particular test that you want to rerun so that you can ensure that the tests are repeatable.

It is generally a good practice to clean up after yourself as well. If a test creates some data, it should also clean it up. However, sometimes, you will want to have a series of tests where data from one test is used in the next. In this case, you might want to create a cleanup section at the end. Setting up these kinds of structures involves sharing variables and data between tests in more complicated ways, so I will wait to go over this until *Chapter 8*, when I discuss workflow testing.

## Summary

This chapter has been filled with ideas on how to create long-lasting and valuable test automation. Over the next few chapters, I will take a lot of the foundations we've laid here and show you how to use various features in Postman. This will help you put the topics from this chapter into practice in your testing.

You have learned the strengths that test automation brings to the table and how to use those strengths to your advantage when creating a test strategy. You have also learned about different types of API tests that you can create, including contract tests, integration tests, endpoint tests, and performance and security tests.

I also showed you how to use Postman to organize and structure variables in ways that will help you understand what an API is doing and what you might need to do if tests fail. Variables are an important part of testing in Postman and so I showed you how to use them and where to store them. You now know which scope to use for different kinds of variables and understand how Postman will resolve variable values if there are conflicts between those scopes.

In addition, you have learned the importance of logging and test reporting when it comes to creating API tests. I gave you a few exercises to work on, but the truth is that this information might still feel a little abstract at this point, and that is okay. These are important principles to understand. I will build on them and make them much more concrete as we progress through this book. You will see how important these foundations are and how to use them in practice as you learn more about Postman.

In the next chapter, we will look at some concrete ways that you can use Postman to do authorization in APIs. We will see how authorization works in APIs, as well as how to set up and use various authorization options within Postman itself.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.com/invite/nEN6EBYPq9>





# 5

## Understanding Authorization Options

In 2017, Equifax, a large credit reporting agency, announced that they had been hacked. Data from 147 million of their users had been stolen. Years of lawsuits and fines followed and by the time everything had been paid out, the hack had cost Equifax more than 1.7 billion dollars.

Although this is one of the most expensive hacks in history, it is far from the only one. Many thousands of companies have been hacked and lost data. The cost of these hacks might range from dollar amounts that end in billions, to those that are “only” in the millions, but the fact remains that security is an extremely important part of any application.

One of the most common ways that attackers get into systems is through APIs. In the Equifax case, the attackers got in initially due to an unpatched server, but then, they were able to extract data for several months by using the APIs that Equifax provides. APIs are meant to be interacted with programmatically. That’s the whole reason we make them, and it allows for all kinds of powerful interactions. However, it also means that nefarious actors can use their power too. Security is important at every level in an application, but APIs are one of the most important places to think very carefully about it.

In this chapter, I will show you some of the strategies that can be used to make APIs more secure. I will also show you some of the things you can do to test for security issues. An important factor of any security system is being able to control who has access to the system and what things they can do. Many APIs have authorization options that give control over these things. These options are obviously important, but they can create some challenges when testing an API. They add additional testing considerations since you need to make sure that they correctly restrict things.



It is important to test for those kinds of things, but authorization can also make it harder to do other API tests. Even if you are not trying to test the authorization options in an API, you will still need to interact with those options when doing other tests on that API.

There are a lot of different ways to set up authorization in an API, and Postman has tools that help you work with APIs that have various types of authorization options. I will spend a good portion of this chapter showing you how to use those tools so that you can effectively test your APIs. In this chapter, I will cover the following topics:

- Understanding API security
- API security in Postman

By the end of this chapter, you will have a good grasp of a lot of concepts related to API authorization. This will include things such as the following:

- The ability to use the various authorization options in Postman to let you access API calls that require authentication to use them
- Authorization and authentication and their use in API security
- Skills that will allow you to deal with various kinds of API authorization
- Using secured API calls regardless of the type of authorization they use
- Being able to set up calls using OAuth 2.0 and other workflows that grant authority implicitly

## Understanding API security

API security is an important topic. This section will introduce some of the basic concepts and terminology used in API security. Later sections in this chapter will walk you through the various ways to authorize an API. However, before I show you how to use those, we need to first talk about what authorization even is. I have been using the term authorization, but the reality is securing an API (or a website) involves two things. It involves **authorization** and **authentication**. These are important topics that underpin all security testing. Although they are often used interchangeably, understanding the distinction between them will help you to effectively test APIs with these options. In this section, we will explore what these two concepts are and how they relate to each other.

## Authorization in APIs

Authorization is how we determine what things a given user is allowed to do. So, for example, if you imagine an online learning platform, you might have different sets of permissions for different users. You might have some users that are students and can only see data in certain ways. A student user might only be able to see a grade mark, while another user who is a teacher might be able to actually modify the grade. There are many additional ways that you might want to authorize a user. If we continue thinking about an online learning platform, the teacher might only be able to modify grades for courses that they teach, and students should only be able to view their own grades and not the grades of their classmates.

Authorization is how we determine which things you have been given permission (are authorized) to do. In an API, this can take the form of determining whether you are authorized to use different endpoints or certain methods on an endpoint. For instance, some users might not be allowed to use a certain endpoint at all, while in other cases a user may be allowed to use an endpoint to GET data, but not to DELETE or modify the data. There may also be cases where an API might return different information depending on the kind of user that you are. An API call that returns data about a user's grades might return information about the actual grade along with some public feedback if you call it as a student, but it might return some additional private comments if you call it as a teacher.

Authorization is a very important aspect of API security. Just because an API requires a username and password to use it, does not necessarily mean that it is well secured. You still need to test that different types of users only have access to the things that they should. In APIs, you especially need to be careful of the way that different relationships and pieces of the API work in revealing information. I have seen a case where an API blocked direct access to certain data for a type of user. However, as that same user, I could see the data if I viewed an object that was a hierarchical parent of the object I wasn't supposed to see. There are often many different paths to the same data, and you want to think carefully about what those paths are when you are testing that the authorization is working correctly.

## Authentication in APIs

You have seen that authorization is about what things you are allowed to do given that you are a certain kind of user. Authentication on the other hand is about determining whether you really are that kind of user. In other words, are you who you say you are?

This might be clearer if you think of a physical authentication/authorization system. Think of a spy movie or a heist movie. In these kinds of movies, the characters need to somehow get into a place they aren't allowed to go. They don't have authorization to go there. Maybe they want to break into a bank vault and steal some valuable jewels. There are certain users that are authorized to go into that bank vault. Maybe the bank manager and the owner of those jewels can go into the vault, but most types of users do not have the proper authorization to go in there.

However, how does the system know which type of user you are? How does it know if you are the owner of the vault or a thief trying to steal the precious jewels? This is where authentication comes into play. You might have to enter in a combination on the lock, or maybe scan your fingerprint or retina to prove to the system that you are the owner and not a criminal. Authentication is about proving that you are who you say you are. Authorization is about making sure that you can only access the things you are allowed to access. Once you have scanned your fingerprint, the system knows who you are – it has authenticated you. The security system then needs to only allow you access to the security deposit box that the user with that fingerprint is authorized to use.

If we take this back to the idea of API authorization and authentication, authentication involves putting in a password or providing a key that proves that you are who you say you are. Once you have done that, authorization determines which data you can see and interact with.

Both of these pieces are crucial to a properly working security system. The two concepts often get conflated, and you will hear one term or the other used to describe them both. This is fine as long as you are aware that there are these two different aspects at play. We need to have some way to verify that someone is who they say that are (a password, for example), but then we also need a set of rules or permissions that allow that authenticated user to access the correct set of resources.

Now that you have the big-picture view of security, let's look at how to deal with security in Postman.

## API security in Postman

Postman has a lot of built-in options for dealing with API security, and in this section, I will show you how to use each of them. When I was getting started with API testing, I found that figuring out how to authorize and authenticate myself was one of the hardest parts of using an API. This section will help you figure out how to handle API security on any APIs you are testing. In the previous section, I talked about the distinction between authorization and authentication. However, as I mentioned, sometimes those two terms are conflated. Postman uses the terminology of “authorization” to combine both concepts, so as I show you how to do this, I will generally stick to using that term as an umbrella term for both authentication and authorization.

In this section, you will learn how to use the various authorization options in Postman. I will go over some common ones such as bearer tokens, and also walk you through the details of how to use OAuth 2.0. In addition, you will learn how to use many of the less common authorization options as well, so that you will be ready to test your API regardless of the type of authorization it uses.

## Getting started with authorization in Postman

Authorization details can be set directly on the **Authorization** tab of a request. In order to do that, do the following:

1. Navigate to a request through the **Collections** tab in the navigation panel.
2. Open the request by clicking on it and you will see an **Authorization** tab.
3. Click on that and you will see a dropdown where you can specify the type of authorization that your API uses.

If you click on that dropdown, you will see that there are several options available.

In this section, I will show you how to set up some of the more common ones from this list. If you need more details on some of the other authorization options, you can find more information in the Postman documentation (<https://learning.postman.com/docs/sending-requests/authorization/>). As you can see in the following screenshot, Postman has many authorization options:

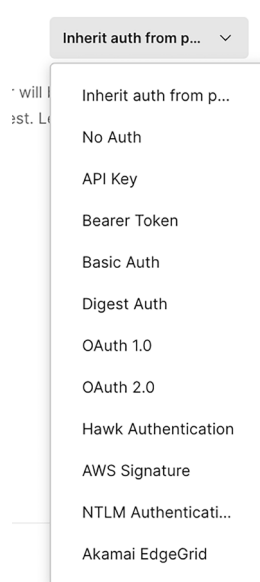


Figure 5.1: Authorization options in Postman

The first option on this list is the **Inherit auth from parent** option. This is a powerful option that makes it much easier to manage an API. Usually, when an API has authorization enabled, it will be required for every endpoint in the API. This means that any time you want to access a request in the API you will need to specify auth options. However, you aren't trying to test the authorization itself; you merely need to enter it so that you can do some other testing. In this case, it makes a lot of sense to only specify the login information in one place rather than repeating it for each request. Postman enables this by allowing you to specify authorization at the folder or collection level. If you click on the **View more actions** menu beside the collection in which your request is located and choose the **Edit** option, you will see that there is an **Authorization** tab. You can set up authorization options here, and then for any request in that collection, you can choose the **Inherit auth from parent** option to use the values you entered in the collection.

If you ever need to update your password or other login information, you can just do this in one spot and all the requests that use it will be updated. Sometimes in API testing, we want to check that different users are correctly authorized and so we might want to use different credentials for some of the API calls. To do this in Postman, you could create a folder inside your collection for each user, and in the options for that folder, you can specify authorization options that can be inherited by the requests in the folder. These options are set on a folder in essentially the same way they are on a collection.

So now that you understand how to structure, organize, and set up authorization options in Postman, let's look at some of the specific options and see how to use them.

## Using Basic Auth

One of the simpler options to understand and use is the **Basic Auth** option, so we will start authentication with that one. This option is used for APIs that require you to specify a username and password in order to use the API. It is probably the simplest form of API authorization, but it isn't one that is generally recommended or used in production systems. The reality is that **Basic Auth** is just a little bit too, well, basic for most uses. However, it is helpful for getting a grasp of how API security works. It is intuitive since it maps well to the way you usually log in to a site with a username and password.

I will use the Postman Echo API, which is a service that you can use to test some different kinds of API calls. The endpoint (<https://postman-echo.com/basic-auth>) is set up to demonstrate the use of basic authorization. In order to set up basic authorization on this endpoint, follow these steps:

1. Create a collection in Postman for the Postman Echo API and add a request to it with the **Basic Auth** endpoint mentioned previously.
2. Click on **Send** and you will get back a **401 Unauthorized** code.
3. Click on the **Authorization** tab and, from the **Type** dropdown, select the **Basic Auth** option.

Postman will display spots for you to put in a username and password.

4. For this endpoint, put in `postman` for the username and `password` for the password.

If you click **Send** again, you should now get back a **200 OK** response and a JSON body like this:

```
{
  "authenticated": true
}
```

Postman makes it straightforward to set up **Basic Auth** like this, but let's look at what is going on under the hood. If you click on the **Headers** tab (note that you may need to click on the eye icon to show the hidden autogenerated headers), you will see that one of the headers Postman has automatically added to this request is the *Authorization* header. The value for this header starts with the keyword **Basic** and this is how the server knows that you are using the basic authorization for this request. The rest of the value for the authorization header represents a base64 encoding of the username and password that you put in.

It is important to note here that base64 encoding is not secure. If you copy the characters from this and go to <https://www.base64decode.org/> and decode it, you will see that it is trivial to get back the username and password from this request. Since base64 encoding is not encryption, it is important to be careful about how you use it. If you are not using a secure (**HTTPS**) connection, this kind of encoding is open to man-in-the-middle attacks and so you want to be careful about using it. Most APIs don't use Basic Auth and instead rely on more complex authorization schemes.

**NOTE:**

The point of testing is to help improve the quality of our product, but unfortunately, it can sometimes cause problems of its own if you are not careful. One area where testing has caused issues in the past is in leaking login information. Many build systems run tests and these build systems are increasingly done on web-based platforms that may be accessible to others. If you are running tests using platforms such as Travis, GitHub Actions, CircleCI, and others, you will want to be careful about how you store login information.

Even if you are not using build systems like these, Postman has features that allow for the sharing of collections and tests with others. You need to think about what credentials you have and make sure that they are stored and shared in safe ways.

Basic Auth is pretty easy to understand but its simplicity means that it doesn't offer a lot in terms of security. You need to be sure to use secure transport with it. It also slightly increases the risks due to the fact that you are passing along your username and password with every request. It certainly isn't the only, or even the most common way to do API authorization. Let's move on to looking at how to use **bearer tokens** in API authorization.

## Using bearer tokens

Bearer tokens are a common way to set up API authorization. They have some advantages over Basic Auth, since they can contain, within them, not only the concept of authentication, but also authorization. Let's look at an example using GitHub:

1. If you have a GitHub account, log in to it.
2. Under **Settings**, go to **Developer settings**.
3. In there, you can choose the **Personal access tokens** option and generate a token.

Here, you can select permissions for this token. This allows you to control what things the user of this token can do.

This approach provides several advantages over basic authentication since in addition to authenticating that someone is who they say they are, with this token you can also determine what actions they are authorized to do. You can also change or revoke what things the bearer of the token can do and so if the token is ever compromised, it is easier to mitigate the damage. Due to this, these kinds of authorization tokens are very common.

In order to use a token like this in Postman, you just need to select the **Bearer Token** option from the **Type** dropdown, and then type or paste in your token. If you look at the **Headers** tab for that request, you will see that Postman has added an **Authorization** header for you. You can also see that the value for this token starts with *Bearer*, which lets the server know that you are using an API token to authorize your request.

Bearer tokens are an example of a larger class of authorization methods known as **API keys**. Bearer API keys have become so common that Postman includes a separate way to set them up, but there are other ways to use API keys as well.

## Using API keys

API keys can be used in a variety of different ways. In order to see this, do the following:

1. Go to the **Authorization** tab of a request in Postman.
2. Choose the **API Key** option from the **Type** dropdown.

You will be presented with a few options.

3. Click on the **Add to** option, which will give you a drop-down choice between adding your API key to the Header or adding it to the query params.

API keys can also sometimes be added directly to the body of a request. However, if your API does this, you will need to manually set it up since Postman does not support this option. When using the bearer token approach to specifying an API key, the API key is added to the **Authorization** header using a standard approach with a specification that the server needs to follow.

What the **API Key** option in Postman enables is support for APIs that manage their keys in a more custom way. Rather than passing in the API key through the authorization header, they might pass it through a custom header that they define for their API. Or they might require that the API key be passed in as a query parameter in the request itself. Since these are more custom approaches, the exact way to do this will vary from API to API and you will need to look at the documentation to figure out exactly how it works. For example, I have seen APIs that require you to pass in the key through the `x-api-key` header. If you had an API that required the key to be passed in this way, you could fill it out in Postman by setting the **Key** field to `x-api-key` and the **Value** field to have the API key in it. You would also need to make sure that the **Add to** option was set to **Header**. If you do this in Postman, you can then look at the **Headers** tab and see that Postman has automatically added an `x-api-key` header with the API key value to your request.



This raises the question if all Postman is doing here is creating a header that has the name you specified in the **Key** field and the value you put in the **Value** field, why not just directly create that header yourself? It wouldn't be any more work for you to directly specify the header yourself on the **Headers** tab, would it? The reason you would want to use the **API Key** option is that it allows you to more easily manage your credentials. You can use the Postman functionality that allows inheriting credentials. It also makes it easier to find which requests you might need to update if credentials change and makes it easier to ensure that credentials are properly secured.

In a similar way, you could specify the key name and value for a query parameter type of API key authorization if that was what the API required. If you do this on the API key page, Postman will automatically add the key and value to the params for that request.

Basic auth and API keys are relatively easy to understand and use, but some APIs use more advanced forms of authorization. Let's look at some of those now.

## Using AWS Signature

Cloud computing is fully mainstream, and many companies now use cloud computing platforms or are trying to move towards them. In many ways, Amazon led the charge when it came to cloud computing and the **Amazon Web Services (AWS)** ecosystem is well known and widely used. A lot of cloud computing systems and resources can be interacted with via APIs, which need to be authorized. Postman provides an AWS Signature authorization option to help with authorizing AWS APIs. Going into all the details of how to set up and use AWS is beyond the scope of this book, but if you have AWS APIs that you are testing, this section will be very helpful. AWS has created their own customized way of authenticating requests. There is a bunch of complexity to it, but essentially you must create a very specific authorization header. Since there is complexity to it, manually constructing this header is rather annoying. In order to set it up in Postman, all you need is your **accesskey** and **secretkey**. These are a lot like the key and the value for the **API Keys** option.

If you have an AWS account, you can set up different access keys in IAM. I won't go into the details of how this works, but you can pick what services and permissions you want the key to have access to. Once you have created the access key, you can copy the access key ID and the secret access key values to use in Postman. Once again, I will reiterate that you need to be careful not to expose these to anyone. Anyone who has these keys can do anything that you have set the key to do.

When you put in the accesskey and secretkey, Postman will create an **Authorization** header for you that matches the syntax AWS is expecting. If you want, you can view that header on the **Headers** tab, to see just how complicated it is. There are a few advanced options that let you set things such as the **AWS Region**, the service name, and the session token.

By default, Postman will assume that the Region is `us-east-1`, so if the resources or service that you are trying to access is in another Region, you can specify it with the **AWS Region** field. If you don't specify the service name, Postman will assume that the token has *execute-api* permissions. However, often, access keys are scoped to a specific service, and so you will need to enter the name of the service that you are trying to access. If necessary (when you have temporary credentials), you can also specify the session token. Postman uses these options to automatically add the headers and parameters that you need to your AWS calls.

AWS has an enormous ecosystem of services and features, and most of them can be accessed with API calls. I certainly can't get into all the details of using AWS APIs, and in fact, can't even get into all the details of using AWS Signature, but hopefully this will help you get started. If you are new to AWS, I would encourage you to dig into how the IAM service works and talk to others on your team who have worked with it more. With Postman's help, by automatically creating headers for you, you should be able to set up calls to AWS!

## Using OAuth

You've probably heard of OAuth before, but if you are anything like me, you've found it hard to wrap your head around this concept. There are a couple of points of confusion that, once cleared up, helped me to get a better grasp of what exactly is going on with OAuth.

The first thing to know is that the OAuth specification, while of course being about authorization, is primarily about the delegation of that authorization. What does that mean? Well, imagine that you are checking into a hotel. The whole point of being there is that you want to get into one of the rooms. Now, the hotel has a key with the ability to open any of those rooms, but they understandably do not want to give you that key. Instead, you go to the front desk and request access to a specific room. The clerk verifies that you can have access to that room, perhaps by swiping your credit card, and then hands you a key card that you can use to get into that room.

Let's walk through the steps for this and then compare those steps to how an OAuth flow works. The first step is that you come into the hotel and give your credit card to the clerk while requesting access to a room.

The clerk then swipes your credit card to determine that you have enough funds available. In this way, the clerk gets approval to issue you a room key and so they hand you a key card.

You can now use that key card to access your room. In this way, the hotel owner can give you access to something without needing to issue you the key to the whole hotel.

They can also easily limit what you have access to and only let you into that one room and maybe the pool and/or weight room, while not letting you into the other rooms in the hotel. This is similar to what is going on in an OAuth flow.

Let's change the terminology to see how it correlates. Now, instead of imagining yourself trying to get access to a room, imagine you want to play a game that needs access to some of your user data from Facebook. In the hotel scenario, you requested access to a hotel room, and in this scenario, the game (or, more generically, the application) asks the authorization server for access to the data that it needs.

When you were requesting access to a hotel room, the clerk used your credit card to figure out whether you should be allowed to have the access that you were requesting. In an OAuth workflow, the authorization server will prompt the user to see if the application should be allowed to have the access it wants. If you approve that access, the authorization server will then give the application a token that will give it access to the data it has requested access to.

A hotel gives you a key card that will open your room for you. An authorization server gives the application a token that it can then use to access the data that it needs from the resource server.

As you can see, an OAuth workflow involves a couple of steps and includes some complications, but it has a few benefits as well. It allows a third-party application to access some of your data without that application needing to be trusted with your password. It also has a lot of the same benefits as a bearer token in that you can change or revoke access if anything goes wrong in the future. This type of auth has become very popular. Many of the big tech companies, such as Twitter, Facebook, and Google allow you to use your accounts with them to log in to other applications. OAuth isn't always simple to set up and use, but it is supported by Postman and is a powerful and common authorization option. Let's look at how to use it in Postman.

## Setting up OAuth 2.0 in Postman

The workflow described previously is an OAuth 2 workflow. There are several slightly different flows that can be used with OAuth depending on what kind of application it is. The exact steps that you will need to use will vary a bit from application to application and you will need to follow up with the documentation for your application to figure out exactly what to do, but in order to help you understand, let's walk through an example showing how to set this up. For this example, I will use the Imgur website (<https://imgur.com/>). If you want to follow along with this, you will need to create an account with that site if you don't already have one.

Before setting up an authorized call, let's first try calling one of the Imgur API endpoints without using authorization by following these steps:

1. Create a collection and name it `imgur`.
2. Create a request in that collection. Name it something like `my account images` and set the endpoint to `https://api.imgur.com/3/account/me/images`. This endpoint will give you a list of all the images that belong to your account.
3. Send the request and you will notice that you get back a **401 Unauthorized** error.

In order to successfully call this endpoint, you will need to get an authorization token. To get this token with the Imgur API, you will need to use OAuth. Before you can set up an OAuth login, you will need an application for it to log in to. Let's take a look at how you can do that.

## Registering an application

Imgur makes it quite easy to create an application. Simply go to `https://api.imgur.com/oauth2/addclient` (ensuring that you are logged in) and fill out the information in the form. Name the application something like `My Postman Test Application` and leave the authorization type option to use OAuth 2 with a callback URL.

The callback URL is the spot users of the application you are creating will be redirected to if they successfully authorize. In this case, we don't have an actual callback URL that we want to redirect users to, but we can use a dummy URL that Postman provides for this purpose: `https://www.getpostman.com/oauth2/callback`. Type or paste this URL into the **Authorization callback URL** field and then add in your email and description if you want and click on the **Submit** button to create your Imgur application. At this point, you may want to copy the client ID and secret as we will use them later. Make sure to keep them somewhere safe though.

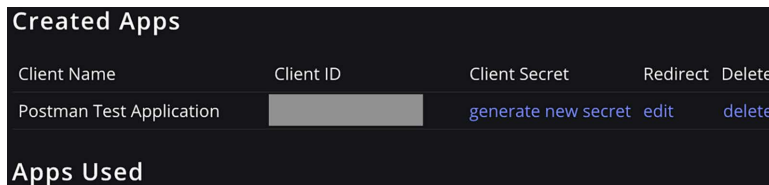
Now that you have an application, we can look at how to set up auth in Postman using OAuth 2.

## Getting an OAuth 2 access token

Go back to the request where you are trying to get all the images that belong to your account and go to the **Authorization** tab. On the **Type** dropdown, select the **OAuth 2.0** option. If you already have a token, you can just enter it in the **Token** field, but we are going to look at how to generate a token when you don't yet have one. As I explained earlier in this chapter, the OAuth workflow requires making a couple of API calls. First, the application you made will need to go to the authorization server and let that server know that it wants a token. This server will check with the user and make sure they are OK with giving the application a token, and if they are, it will return the token and redirect to the callback URL.

These steps can be performed manually, but Postman has set it up so that it can do some of this automatically in the background for you. In order to use that functionality, scroll down to the **Configure New Token** section. This will give you a form with several fields to fill out:

- First of all, give the token a name. Postman will store the token for you so that you can use it in other requests if you want.
- Ensure that the **Grant Type** option is set to **Authorization Code**.
- **Callback URL** needs to be the exact same as the one the application will redirect to. So, in this case, `https://www.getpostman.com/oauth2/callback`.
- **Auth URL** is the API endpoint that you call in order to ask the authorization server if you are allowed to get a token. When you call this URL, the user will be prompted to make sure they are OK with your application having access to their data. For the Imgur API, that endpoint is `https://api.imgur.com/oauth2/authorize`.
- Once your application has received authorization to proceed, it gets a very short-lived key that it can exchange for a more permanent token. It does this with the **Access Token URL**. The Imgur API endpoint for this is `https://api.imgur.com/oauth2/token`.
- In order to authorize your application, the authorization server needs to know what application it is being asked to authorize. This is where the **Client ID** and **Client Secret** fields come in. If you didn't copy them when you created your application, you can get them by going to `imgur.com` in your browser, clicking on your username, and then selecting **Settings**. On the settings page, choose the **Applications** option. You should now be able to see the **Client ID** for your application, and you can click on the **generate a new secret** option to get the **Client Secret**, which is shown in the following figure:



Client Name	Client ID	Client Secret	Redirect	Delete
Postman Test Application		<a href="#">generate new secret</a>	<a href="#">edit</a>	<a href="#">delete</a>

Apps Used

Figure 5.2: Getting the client ID and client secret from Imgur

- The **Scope** field is used if the token should have limited abilities and **State** is an additional security measure that some APIs have. You don't need to worry about either of those with the Imgur API.
- For the **Client Authentication** option, ensure that it is set to use the **Send as Basic Auth header** option.

Whew! That's a lot of information to fill out, but once you have filled out all of those fields, the form should look something like this:

**Configure New Token**

Configuration Options ● Advanced Options

Token Name

Grant Type

Callback URL ⓘ

☐ Authorize using browser

Auth URL ⓘ

Access Token URL ⓘ

Client ID ⓘ

Client Secret ⓘ

Scope ⓘ

State ⓘ

Client Authentication

ⓘ

Figure 5.3: Settings for getting an OAuth 2.0 access token in Postman

Notice that Postman has a warning suggesting that you use variables instead to keep your sensitive data secure. I know I'm harping on about it in this chapter, but once again I will remind you to make sure to save and use credentials like this in a secure way. You should now be able to click on the **Get New Access Token** button. If you have filled out all the information correctly, you will get a popup asking you if you want to allow your application access to your Imgur data. Enter your username and password if necessary and then click **Allow**. If everything is successful, you will see a popup with the details of your access token. You can click on **Use Token** to tell Postman to add the access token to the appropriate field.

If you scroll up, you should see that the token has been added to **Bearer Token** field and that the name of the token in the dropdown is set to the name you gave your token. You can now use this token in other requests by choosing the OAuth 2.0 authorization type and then picking the token from the available tokens dropdown.

Now, if you send your request again, you should get back information about all the images that are associated with your account. If you just created your account, you might not have any images, but you should still get back an empty list and the call should return a 200 OK status code.

As you can see, there is a lot of complexity involved in using OAuth for API authorization. Postman does give you some help with it though. I've shown you how to set it up for one API, and that should give you a pretty good idea of how to do it for others. Each API, of course, is going to have different settings for things like the access token or Auth URLs and so you are going to need to figure out the specific values of these for the API you want to test. These things will usually be in API documentation. You will also need to figure out the client ID and secret, which might be things you need to ask others about or get access to internally. Note that there are some other options that might differ between APIs. For example, the Imgur API uses an authorization code for **Grant Type**, but other APIs might use **Client Credentials** or other grant types. Once again, you will have to look in the documentation of the API you are using to get information on which specific settings to use, but once you have all that information, you should be able to set up an access token in Postman.

## OAuth 1.0

So far in this chapter, I have been using the terms OAuth and OAuth 2.0 interchangeably. Almost all APIs that use OAuth now use OAuth 2.0. Despite the similar names, OAuth 1.0 is quite different from OAuth 2.0. It does have some similar ideas and laid a lot of the groundwork for the kinds of flows that are used in OAuth 2.0 authorization. However, OAuth 1.0 had several limitations that were made much better in OAuth 2.0 and so OAuth 1.0 is not used much anymore.

Postman does still support this option, so if you are testing an API that uses it, you can do so in Postman. However, since it isn't used much in industry, I won't go through the details of setting this up. If you do need that information, you can look at the Postman documentation to help you get started with it (<https://learning.postman.com/docs/sending-requests/authorization/#oauth-10>).

## Digest authentication

Postman provides authorization options for setting up **Digest** and **Hawk** auth. These are less common authorization methods. The Hawk authentication scheme was meant to solve similar problems to the OAuth 2.0 standard, but it never saw widespread adoption, so although it is still used in some APIs, most new APIs will use OAuth 2.0 instead.

Digest authentication helps to make it more secure to use a username and password if you're sending your data over an unencrypted connection. Although it is more secure than just sending a plaintext password, it still has its downfalls and since most sites now use encrypted connections, it doesn't have much value in the modern API ecosystem.

These are not very common authentication systems, but it is still worth spending a few minutes understanding how they work. Understanding them will also be helpful for broadening your overall understanding of how authentication and authorization work in APIs. Postman provides a digest auth method in the postman-echo API that we can use to gain an understanding of how this auth method works.

The main purpose of digest auth is to allow you to send your password to the server in an encrypted manner. In order to do this, you actually need to send two requests to the server. The first time you send the request, the server will respond with a few values. It will send back details about the realm you are attempting to access and a nonce, which is basically a random set of characters you can add to your password to make it harder to decrypt. Once you have all that information, you need to put it all (username, password, nonce realm, etc.) into a hashing algorithm like MD5. This will turn it into a random set of characters that you then send back to the server. The server will create its own hash using the same algorithm (and using the password it has stored locally) and compare it to the one you sent to see if you are authorized to access the resource you are requesting.

Let's take a look at this in Postman:

1. Add a request called **Digest Auth** to the **Postman Echo** collection.
2. Set the **URL** of it to `https://postman-echo.com/digest-auth`.
3. On the **Authorization** tab, choose **Digest Auth** from the **Type** dropdown.
4. Enter the username and password. In this case, the username is `postman` and the password is `password`.



- 5. Under the dropdown where you selected **Digest Auth**, there is a checkbox that you can select to disable retrying the request. By default, Postman will send the first request, extract the necessary information from it, and immediately send the second request to authorize you. In order to better understand how this works, let's disable this request.

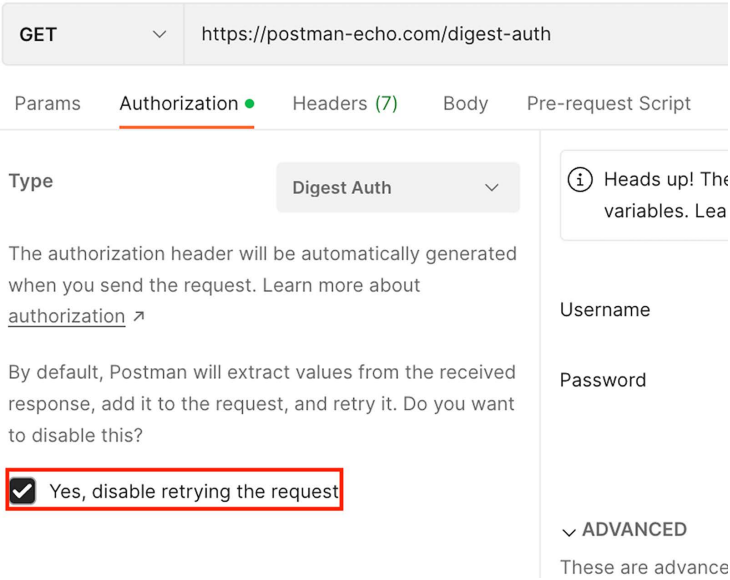


Figure 5.4: Disable retrying Digest Auth

- 6. Send the request and you will get back a **401 Unauthorized** response. If you look in the headers of the response, you will see that it has a **WWW-Authenticate** header with **Digest Realm** and **Nonce**. You can copy these options into the **Advanced** section of the auth settings.

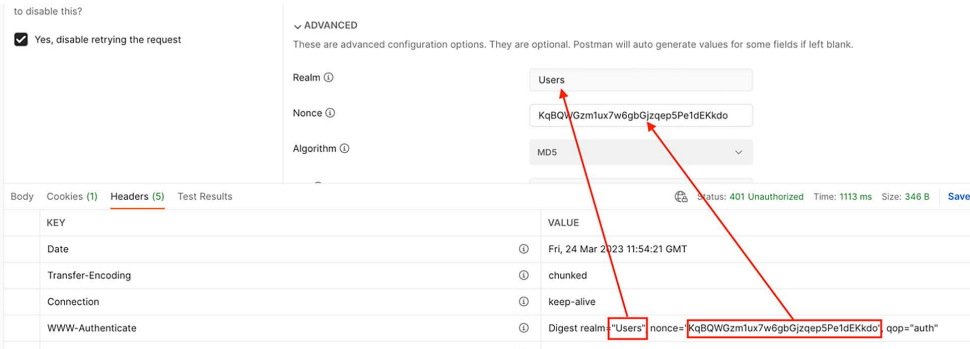


Figure 5.5: Copy auth options

If you send the request again, you will get back a 200 OK response showing that you are authenticated. If you uncheck the option to disable retrying the request and remove the **Realm** and **Nonce** values that you put in, you can send the request again and see that Postman seamlessly sends both requests for you in the background.

## Hawk authentication

Hawk is like **Digest Auth** in that it sends an encrypted hash of the password (or key in this case) over the network, rather than sending plain text passwords. Both of these auth methods make it possible to send passwords to a server that is using a non-secure (HTTP) rather than secure (HTTPS) connection. Digest auth requires two requests to the server for each auth session since that server has to tell the client the nonce so that the client can use it to encrypt the password. In the case of **Hawk Auth**, the client generates its own nonce and sends it to the server, meaning that it only needs to send one request to do the authorization.

Postman has an endpoint in their echo API that can be used to test this:

1. Create a request in Postman and set the URL to `https://postman-echo.com/auth/hawk`.
2. On the **Authorization** tab choose the **Hawk Authentication** option.
3. Set **Hawk Auth ID** to `dh37fgj492je`.
4. Set **Hawk Auth Key** to `werxhqb98rpaxn39848xrunpaw3489ruxnpa98w4rxn`.
5. Send the request.

If you look on the **Headers** tab of the request, you can see that Postman has added an **Authorization** header with a few different details.

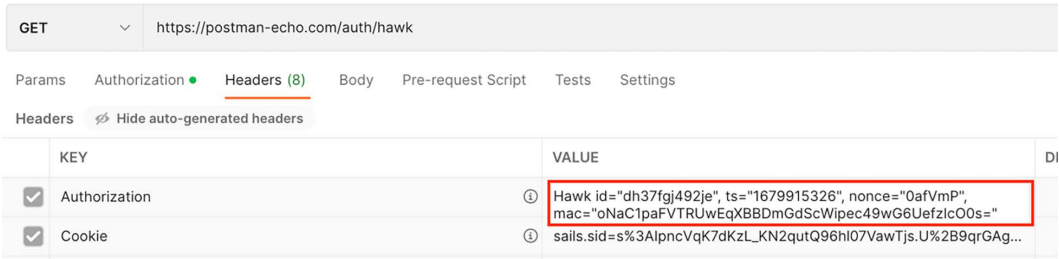


Figure 5.6: Hawk auth headers

It sends the Hawk ID that you specified, and then it also sends something called **ts**. This stands for **Time Stamp** and is used by the server to determine how long the request is valid.

Since Hawk requests are sometimes sent over unencrypted connections, it is possible that someone could intercept the request and could try to “replay” it, meaning they send it again at a later point to try and get access to the server. If they tried to do that, the server would notice that the timestamp was from too long ago and would reject the request. It also sends a nonce that the server uses together with the timestamp and the copy of the key that it has locally to calculate the hash. The last item (mac) is an encrypted hash of the password.

With the Postman echo API, the hawk Id and key and the type of encryption method to use are all posted publicly since it is meant for testing and not for true security purposes. If you are testing a real-life API that uses **Hawk Auth**, you will need to find out from your team how to get the ID and keys that you need to access the API.

## Using NTLM authentication

NTLM stands for **New Technology LAN Manager**, and it is a set of security protocols that Windows uses to provide authentication to users. It can be used in APIs to allow users access to resources on the API based on their current Windows login information. If you are working with an API that has this kind of authentication set up, you can choose that option from the **Authorization Type** dropdown and then enter your Windows username and password.

### A WARNING ABOUT PASSWORDS



Any time that you use login credentials, you need to be careful about sharing collections or requests. This is especially important in this case, since you are using the login credentials to your computer and probably many other Microsoft accounts. Make sure to protect them carefully!

Postman will default to sending the request twice, since the first time it will get back the security values that it needs to be successful on the second try. A lot of APIs that use this kind of authentication are internal APIs, so if you need more information on setting it up for a particular API that you are working on, you may need to talk to the internal stakeholders at your company to figure out the details.

## Using Akamai EdgeGrid

Akamai Technologies is a global cloud services company. They have a variety of services and many of those services are supported by APIs. They also have their own somewhat unique way of authenticating applications within their network and have created an authorization helper that you can use in Postman. The details of this are beyond the scope of this book, but if you need more information, you can check out the Akamai developer documentation ([https://developer.akamai.com/legacy/introduction/Prov\\_Creds.html](https://developer.akamai.com/legacy/introduction/Prov_Creds.html)).

Some of the options for authorization in Postman are more common than others. Most of this chapter has focused on how to set up credentials so that you can log in to an API and make the calls that you want. This is an important part of being able to test an API, but I also want you to think about why we need all these ways of authorizing and authenticating APIs in the first place. Why do we even have all the options in the API ecosystem? The reason, of course, is that there are those who want to use APIs in ways that would cause harm.

Postman has a lot of different authorization options. You may not have needed to carefully study each of them, but hopefully, whichever option your API uses, you will be able to get started with it. I've shown you how to use Basic Auth, bearer tokens and API keys, and authentication for AWS APIs. You also now understand how OAuth workflows work and how you can use them in Postman. In addition, I introduced you to some lesser-known authorization options such as Digest, Hawk, NTLM, and Akamai EdgeGrid. You certainly won't encounter all of these options in one API. In fact, you might not even encounter some of them in the course of an entire career, but you now have the foundational knowledge that you need so that you can log in to any API you face.

Knowing how to log in to an API so that you can make the calls you want is important, but don't forget that APIs are one of the most common places that attackers target when they are trying to break into a system. In *Chapter 14*, we will go into more detail on security testing, but even if you don't ever need to do security testing, you can still help keep your application secure by protecting the credentials that you have. I have mentioned this a couple of times already, so before I close this chapter, let's look at how you can safely use credentials in Postman.

## Handling credentials in Postman safely

Postman recommends using variables to store sensitive data. You can set a variable to be **secret** and Postman will hide the value for you. However, that functionality only works for environment and global variables. So, if you are trying to protect credentials, you will need to save them as variables in one of those contexts. When you create a global or environment variable, you have the option to set the type to **secret**.

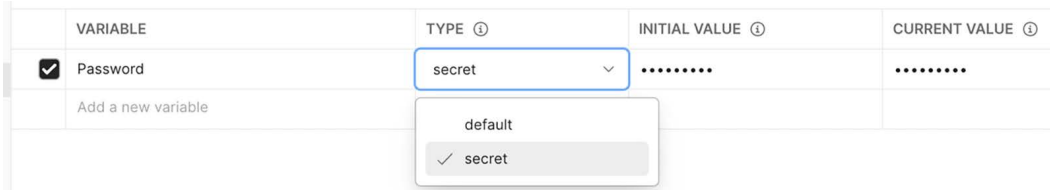


Figure 5.7: Saving secret variables

Since environments can be shared with others, only those with editor access on the environment can change the variable type. If you have collaborator permissions, you can view the value of a secret variable by selecting the eye icon that appears when you hover over the **VARIABLE** value.

## Summary

This chapter has covered a lot of territory. Security is a complex and important topic and understanding how to work with it is an important part of API testing. In this chapter, I have shown you how to think about API security and what the distinction is between authorization and authentication in security. I also showed you how to use the various Postman authorization types to give you access to secured APIs. You learned how to log in with many different authorization types, ranging from Basic Auth to API keys and tokens, to OAuth 2.0. I also showed you some of the other authorization options in Postman and showed you how to get started with them.

I hope you are excited to continue learning about Postman in the next chapter, where I will show you how to create test validation scripts. This will involve using JavaScript in Postman and will be a lot of fun as you learn how to check that requests are doing what they should be and a lot of other powerful things that Postman can help you with. Let's continue with that in the next chapter!

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.com/invite/nEN6EBYPq9>





# 6

## Creating Test Validation Scripts

At one company that I worked at, I was trying to figure out the value of some of the test automation scripts that we had. In order to do this, I was analyzing the test results to see which scripts were giving us the most information. One of the rules I used to determine which scripts might not be adding value was to look at scripts that had never failed. My hypothesis was that if a script had been running for some time and had never failed, it was unlikely to fail in the future and so was not giving us valuable information. I had identified several test scripts that had never failed and was looking through them. Imagine my surprise when in several of them I found assertions that were checking things such as whether `true==true` or `5 == 5`? No wonder the tests had never failed. It was impossible for them to fail.

Although these were egregious examples, the reality is that often a well-designed test suite will fail to deliver on its promise because of poor assertions. An **assertion** is the part of a test where you check if the results are what you expect them to be. You can have a test suite that checks all the necessary endpoints with all the correct inputs. It can have perfect coverage and be impeccably structured, but without good assertions, it isn't doing you much good.

In this chapter, I will show you how to set up good test validation scripts in Postman. Postman uses JavaScript for this. If you are not familiar with JavaScript, don't worry about it. I will walk carefully through the examples so you should be able to follow along. Postman also provides some helpful examples that you can use. You do not need to be an expert in JavaScript in order to follow along in this chapter, but I hope that by the end of this chapter, you will be an expert in creating good test validation.



In addition to using JavaScript for test validation, Postman provides ways to create **setup** and **teardown** scripts. You can use them to set some things up before you run a test and to do cleanup after a test has been completed. By the end of this chapter, you will be able to use all this functionality in Postman. You will be able to create test validation assertions using JavaScript in Postman, validate body and header data in API responses, use the assertions that the Postman team has created, set up variables and other data before sending a request, create workflows that include multiple requests, create loops to run the same request multiple times with different data, run requests in the collection runner, and use environments to manage and edit variables.

The following are the topics that we are going to cover in this chapter:

- Checking API responses
- Setting up pre-request scripts
- Using environments in Postman

## Technical requirements

The code used in this chapter can be found at <https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/master/Chapter06>.

## Checking API responses

Since Postman uses JavaScript for the checks in a test, it has a lot of power and flexibility built into it. I'm going to walk you through various things that you can do with this. In order to do that, it will be easiest to work with actual API calls. For that purpose, I will once again use the Star Wars API (<https://swapi.dev>). If you don't have one yet, create a collection in Postman called something like `Star Wars API - Chapter 6`, and in that collection, create a request called `Get First Person`. This request should call the `/people/1` endpoint from the Star Wars API. You can also download the collection from the GitHub repository for this course (<https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/master/Chapter06>) and then import the `Star Wars API_Chapter6_initial.postman_collection.json` file if you prefer.

When I made this collection, I also created a variable called `base_url` that specifies the base URL for the `swapi.dev` service. I will be creating a few different requests as examples in this chapter, so it will be helpful to have that variable. If you set up the collection on your own, you will need to edit the collection and add the variable, giving it a value of `https://swapi.dev/api`.

If you imported the collection from GitHub, the variable should already be there for you, but in either case, you will need to go to the `Get People` request and modify the URL so that it references the variable and starts with `{{base_url}}`. Once you have that all set up, go ahead and send the request.

Being able to send a request like this is great, but if you automate it, how will you know in the future if it is working correctly? In this section, I will show you how to add checks that can verify this. I will also show you how to check data in API responses and headers and how to use some of the provided assertions so that you can check anything that you need to. Status codes are the easiest way to see at a glance if something might have gone wrong with an API request and so they are the first thing that we will look at checking.

## Checking the status code in a response

In Postman tests are added in the **Post-response** area, which you can find on the **Scripts** tab. You can find this tab on each request. Go to the **Scripts** tab for the request you made and click on the **Post-response** section and, as you can see in the following figure, the right-hand panel has several different snippets available for you to use:

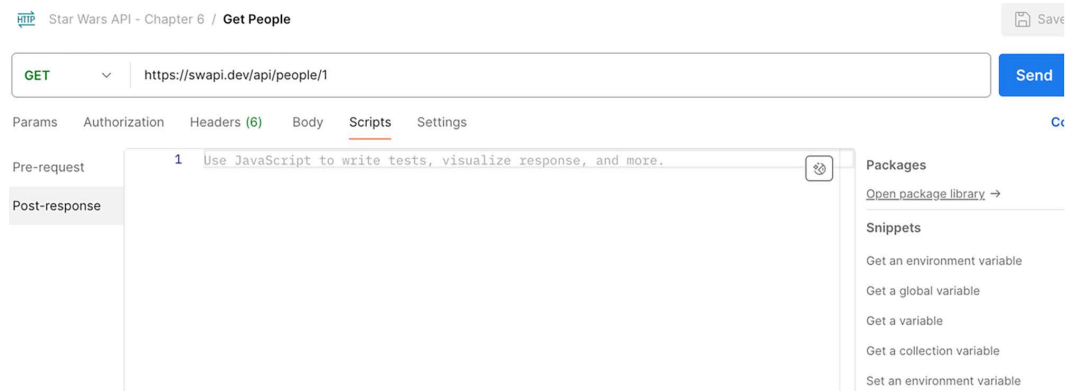
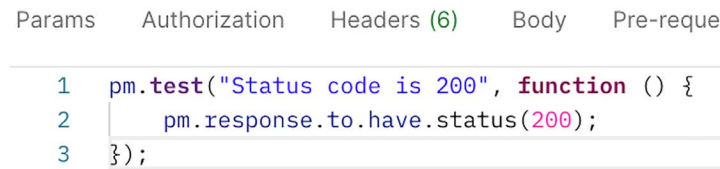


Figure 6.1: Test snippets

These snippets are little bits of JavaScript code that Postman provides to make it easy to add some common checks to your tests. Scroll down in the list until you see one that says `Status code: Code is 200` and click on it. When you do that, you will see that Postman adds some JavaScript code into the test panel.

You should see code that looks something like the following figure:



The image shows a snippet of JavaScript code in Postman's test editor. The tabs at the top are Params, Authorization, Headers (6), Body, and Pre-request. The code is as follows:

```
1 pm.test("Status code is 200", function () {  
2     pm.response.to.have.status(200);  
3 });
```

Figure 6.2: Snippet checking that the status code is 200

Now, if you send the request, this code will check that the request returns a status code of 200. If the request returns any other status code, this check will fail. You can check on the results of the tests by looking at the **Test Results** tab in the bottom panel:

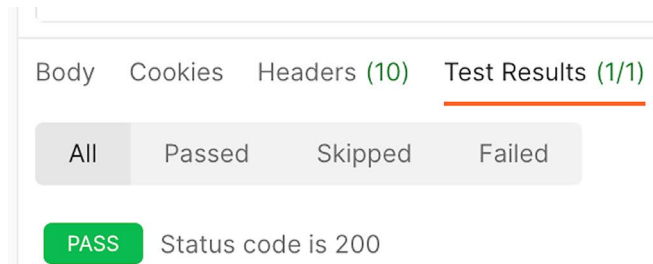


Figure 6.3: Test Results

If you are not familiar with JavaScript, the code for this test might be a bit intimidating, so let's work through this check and see what is going on.

## Using the `pm.test` method

In the first place, we have the `pm.test` function. The `pm` object is a JavaScript object that Postman provides that gives you access to data from your requests and their responses. It also lets you access and manipulate variables and cookies. In fact, most of the JavaScript functionality that you will use in Postman is provided through the `pm` object.

The test method is an asynchronous method that you can use to set up each check that you want to do. It takes in two arguments. The first one is the name of the test as a string, in this case, "Status code is 200". Note that enclosing text in double (or single) quotes means that it is a string. This name will appear in the test result output, so you will want to make sure that it accurately describes what the test is doing so that you can better understand your test results.

The second argument needs to be a function that does the actual check that you are interested in. JavaScript provides a lot of shortcuts for this kind of stuff, so you can define your function directly in the argument to the test method. In this case, the snippet is using an unnamed (or anonymous) function. This is the only spot this function is being used so it does not need a name. The function is defined by `function ()`, which just means that you are creating an anonymous function. Since there is nothing in the brackets, we know that this function does not take in any arguments. However, the `pm.test` method does expect the function to return a Boolean (a value that is either `true` or `false`). It should not return a string, or an integer, or anything like that. You can do whatever checks you want inside the function, but you need to ensure that it returns a Boolean value.

The actual work that the function does is then defined inside the curly braces. So in this case the function is merely calling `pm.response.to.have.status(200);`. Tests need to take in a function like this because they run asynchronously. This means that JavaScript (and hence Postman) will not wait for one test to finish before it goes on to the next one. In other words, if you have two tests set up for a request and the first test is doing something that takes a while to process, that test will start, and then while Postman waits for it to finish, it will go ahead and start the next test. This is why we need to give the test a function. Essentially, the test will start, and then once it has the data that it needs, it will call the function that we supplied and execute the code in that function. This function that gets called is sometimes called a **callback function**, so if you hear that term, it is just referring to a function that gets passed into another function and that will be called once that function is ready.

## Using Chai assertions in Postman

There is one last piece to this test that we haven't looked at yet. That is the actual code that is being executed. The snippet uses the `pm` object to access the response data. The `pm.response` object contains various information about the response. In this example, you are getting the response and then creating an assertion on that response to check that the response has the status 200.

Assertions in Postman are based on the capabilities of the **Chai Assertion Library**. This is a very common JavaScript assertion library and is used in many JavaScript unit testing frameworks. The Chai framework supports **Test-Driven Development (TDD)** and **Behavior-Driven Development (BDD)** approaches. Postman uses the BDD style of Chai assertion. You can read more about the various assertions on the Chai website (<https://www.chaijs.com/api/bdd/>).

Chai assertions are very nice to read. They match up well with how we would speak about the assertions in natural English. I'm sure that you were able to figure out that when the snippet said `to.have.status(200)`, it meant that we expected the response to have a status of 200. This readability makes it easy to figure out what a test is checking; however, I have found that it can be a bit tricky sometimes to figure out how to craft these assertions. As we go through this chapter, I will show you several different assertions. As you see (and create) more of them, they will get easier to use.

## Try it out

This is a very basic check, but I want you to take a couple of minutes to make sure you understand how it works. Try changing the expected status code and ensure that it fails.

The script is asserting that the response should be 200. Play around with this assertion a bit and make sure you understand what is going on. Can you make it fail? Make sure you understand how it works. I will be showing you some more assertions, but they share many of the same components as this one, so make sure you fully understand what is going on here before moving on to look at how to check data in the body of a response.

## Checking the body of a response

You have seen how to verify that an API response has the status code you expect it to, but there is a lot more that you can do with tests in Postman. You can check many things in the data of the response itself. I'm going to show you a couple of examples of how to do that so that you can have the confidence to create these kinds of assertions on your own.

## Checking whether the response contains a given string

For the first example, I will show you how to check that a response contains a string that you expect it to. The snippets are a very helpful way to learn about the available functionality, so let's use another one to look at how you can verify that the body of a request has the correct data. Scroll down in the snippets section until you see a snippet called `Response body: Contains string`. Ensure that your cursor is on the last line of the Post-response text field and then click on the snippet to add the code for it to the tests.

You should see a code snippet that looks like this:

```
pm.test("Body matches string", function () {  
    pm.expect(pm.response.text()).to.include("string_you_want_to_search");  
});
```

As you can see, there is a lot of similarity between this snippet and the previous snippet. It once again uses the `pm.test` method to define the test and we give it a function that defines an assertion. In this case, the assertion is a bit different, though. You are still looking for response data with the `pm.response` object, but now you have the `.text()` method attached to it. Since the `pm.response` object is just that – an object, you need to turn it into a string before you can check whether it includes the string you want. This is what the `.text()` method does. It turns the response object into a string that Postman can search through. Replace `string_you_want_to_search` with a search term such as `Luke`. Give the test a more descriptive name as well – perhaps something like `Check that the response body contains Luke` – and send the request again. In the **Test Results** tab, you should now see that there are two passing tests.

## Checking JSON properties in the response

Now let's look at another example snippet. This time choose the `Response body: JSON value check` snippet. You will see that the following code is added to the tests:

```
pm.test("Your test name", function () {  
    var jsonData = pm.response.json();  
    pm.expect(jsonData.value).to.eql(100);  
});
```

There are a couple of new things in this snippet. In the first place, you can see the use of the `var` keyword. This is used in JavaScript to define a variable, so in this case, we are creating a variable called `jsonData`. The data for this variable comes from the `pm.response` object, but this time instead of calling `.text()` to convert it into a string, the snippet is calling `.json()`, which will turn it into JSON. **JSON** stands for **JavaScript Object Notation** and it is a format that is used for storing and transporting data. It is structurally similar to a dictionary or hash map object and is a very common standard. Many APIs will return data in a JSON structure. The Star Wars API is no exception to this, so you can directly convert the response object for a call to that API into JSON.

The variable can then be used in an assertion. You can see that Postman has put in a default assertion that expects the value of the `jsonData` variable to equal `100`. If you send the request, you will see that this test fails with an assertion error where it tells you that it expected undefined to deeply equal `100`, as you can see in the following figure:

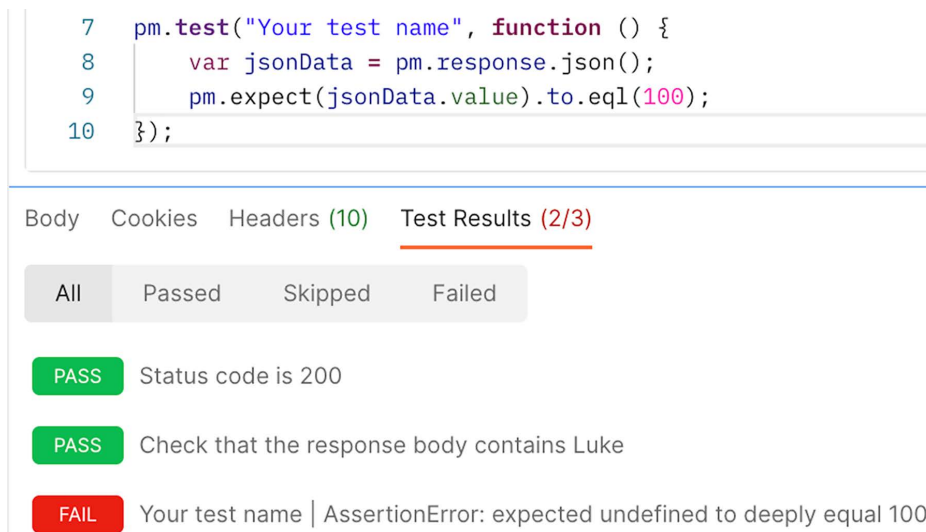


Figure 6.4: Test assertion fails

This warning sounds a bit cryptic, but all it is telling you is that it was trying to compare the two values that you gave and one of them was undefined while the other was `100`. It was expecting those two values to be equal and hence the assertion failed. The “deeply” part of “deeply equal” just means that it is checking every key and value of the JSON object. This error tells you that `jsonData.value` is undefined. But why? Well, in order to understand that, let’s look at the console log.

In JavaScript, you can log things to the console. This is similar to printing in other languages and is very helpful when trying to debug something, so let’s try it out here. Go to the line immediately after the line on which the `jsonData` var is defined and add this code:

```
console.log(jsonData);
```

Now send the request again. In order to see the data that has been logged, you will need to open the console in Postman. As shown in the following figure, you can do that by clicking on the **Console** button at the bottom of the Postman app:

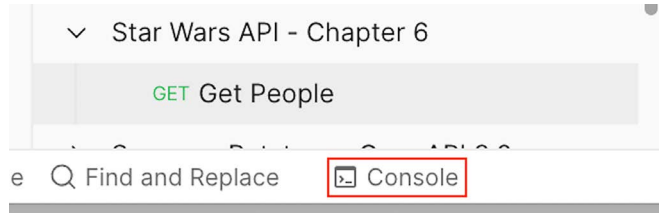


Figure 6.5: Open the Postman console

At the bottom of the console, you should see the `jsonData` object that you logged:

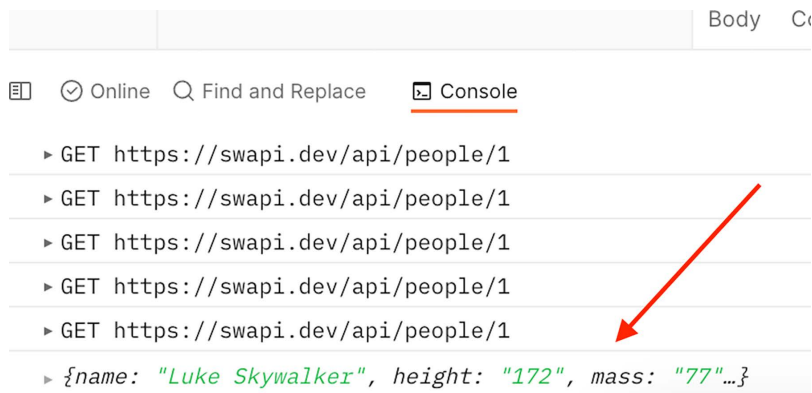


Figure 6.6: Logged object

This looks like what you'd expect to see, so why does the assertion think that things are undefined? Well, we need to look for a minute at the `.value` operator. This snippet is a bit tricky. This looks like it is a valid operator, but it is actually just a placeholder to let you know that you need to get some value out of the `jsonData` object. In essence, this snippet is trying to find a field named `value` in the response. However, the response fields in the case are called `name`, `height`, and `mass`, and so since it can't find one named `value`, the variable is undefined. In order to get the information that you want, you just need to put in the name of one of the fields you are interested in. So to get the height, you can change the code to:

```
jsonData.height
```

You will also need to change what you expect it to equal. In this case, it should be equal to `"172"`, so put that in to make the line look like this:

```
pm.expect(jsonData.height).to.eql("172");
```



Now if you send the request, it should pass. Don't forget to remove `console.log` and change the name to something more descriptive.

**NOTE:**

You can access the properties of objects in JavaScript using either bracket or dot notation. They both give you access to the same information, but I think that dot notation is more readable than bracket notation, so I will stick with that throughout this book. In dot notation, you can access the properties of an object by using the syntax `objectname.propertyName`. You can also access nested objects in this way, and so you could see something like `objectname.subobjectName.propertyName`.

This example should give an idea of the kinds of options you have for checking response data. Many APIs send responses in JSON format, and with the options you have learned about in this example, you should be able to create checks for almost anything that you might get in an API response. In order to make sure that you understand what is going on here, though, I would encourage you to explore this a bit more.

## Try it out

You have seen a couple of examples that show how to check for data in API responses. Now I want you to try your hand at this without step-by-step examples. See if you can take what you have learned and put it into practice on your own. You can play around and set yourself a few challenges on your own if you want. To help you get started, here are a few ideas of tests that you could create:

- Create a test that verifies Luke's eye color.
- Create a test that verifies the URL of Luke's home world.

If you get stuck on these, you can check out my solutions in the GitHub repository for this course (<https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/master/Chapter06>). Don't look at the solutions until you have tried it out on your own though!

Now that you know how to check data in the body of a response, it's time to look at how to check the headers of responses.

## Checking headers

An API response includes a status code, a body, and a header. Sometimes, there are important things in the header that you want to be sure are getting sent back correctly. As you might guess, you can create assertions in Postman that let you check header data. Once again, the easiest thing to do is to start with a snippet—in this case, the `Response header: Content-Type header check` snippet. Choosing that snippet should add this code to the tests:

```
pm.test("Content-Type is present", function () {  
    pm.response.to.have.header("Content-Type");  
});
```

This is similar to the code in the body snippets. The only difference really is that the assertion has the `.header` assertion in it. This is an assertion that is specific to Postman. Postman has a Chai plugin that extends the common Chai assertions to give you access to a few custom ones that they provide. You could build this same assertion without using this custom assertion type, but it would be more complicated to do so.

## Custom assertion objects in Postman

You may not have noticed, but you actually used another custom assertion object in one of the earlier examples that we went through. When checking the status code, the `.status` assertion is a custom assertion type. Postman has also created several other assertions like this that can make it a lot easier to assert on common response objects. I have listed the custom assertion methods that Postman provides, along with examples of how to use them. These assertions are specifically designed to help you verify API responses, so you will probably use them a lot when creating tests in Postman:

- `.statusCode`: This assertion checks whether the numerical status code of the response matches the one you specify. Here's an example usage:

```
pm.response.to.have.statusCode(200);
```

- `.statusCodeClass`: This assertion checks whether the response code is in the right class. Class 2 is a status code that is in the 200s and 3 is one in the 300s, and so on. Here's an example usage:

```
pm.response.to.have.statusCodeClass(2);
```

- `.statusReason`: This assertion checks whether the reason, which is a text response corresponding to the status code, matches what you specify. Here's an example usage:

```
pm.response.to.have.statusReason('OK');
```

- `.status`: This assertion allows you to verify the status by specifying either the status code or the status reason. It is essentially a combination of `.statusCode` and `.statusReason` assertions. Here are two example usages:

```
pm.response.to.have.status('OK');  
pm.response.to.have.status(200);
```

- `.header`: If you only input one argument, this assertion will check for the existence of a header that matches that argument. If you specify two arguments, it will check that the given header (specified by the first argument) has the given value (specified by the second argument). Here are two example usages:

```
pm.response.to.have.header('Content-Type');  
pm.response.to.have.header('Content-Type', 'application/json');
```

- `.withBody`: This assertion checks that the response has a body. Here's an example usage:

```
pm.response.to.be.withBody;
```

- `.json`: This assertion checks whether the body is in JSON format. Here's an example usage:

```
pm.response.to.be.json;
```

- `.body`: This assertion can be used to check whether the response has a body and whether that body contains a given value. You can specify the expected value as a simple string, in which case it will check whether that string is found anywhere in the body. Alternatively, you can specify the expected value as a regular expression or as a JSON key/value pair. In those cases, the assertion will search for a match to your query. Here are some example usages:

```
pm.response.to.have.body;  
pm.response.to.have.body('some text');  
pm.response.to.have.body(<regex>);  
pm.response.to.have.body({key:value});
```

- `.jsonBody`: If no argument is specified, this assertion checks whether the body of the response is in JSON format. Otherwise, it will check for the existence of the given JSON object in the body. Here are some example usages:

```
pm.response.to.have.jsonBody;  
pm.response.to.have.jsonBody({a:1});
```

- `.responseTime`: This assertion checks how long the request took. It can also be used to check whether a response time was above or below a given value or if it is within a given range. The time values are given in milliseconds. Here are some example usages:

```
pm.response.to.have.responseTime(150)  
pm.response.to.have.responseTime.above(150);  
pm.response.to.have.responseTime.below(150);  
pm.response.to.have.responseTime.within(100,150);
```

- `.responseSize`: This assertion checks the size of the response. It can also be used to check whether a response size is above or below a given value or if it is within a given range. The response sizes are given in bytes. Here are some example usages:

```
pm.response.to.have.responseSize(50);  
pm.response.to.have.responseSize.above(50);  
pm.response.to.have.responseSize.below(100);  
pm.response.to.have.responseSize.within(50,100);
```

- `.jsonSchema`: This assertion checks whether the response follows the specified schema. Here's an example usage:

```
pm.response.to.have.jsonSchema(mySchema);
```

These assertions can help you deal with the various responses that you will get when making API requests. This list gives some simple examples to help you get started with using these assertions, but don't stop there. You can use these assertions in combination with assertions provided by the standard Chai library as well. For example, you could add `.not` to the assertions in order to negate any of them. Use these listed examples as starting points for building out your own assertions as you create your tests.

## Creating your own tests

I have shown you several examples using the snippets that Postman has created in order to help you get started with creating tests. These snippets are a great place to start if you don't know much about JavaScript as they give very clear examples of how to do certain things. However, they are just a starting point. Since assertions in Postman are built on JavaScript, there is a lot of flexibility and power in what you can do. You may benefit from learning some basic JavaScript, but even without that you should be able to create a lot of your own assertions merely by using the snippets along with the kinds of commands you have learned about in this section.

### Try it out

As I keep repeating in this book, the best way to learn is by doing, so try creating a few of your own assertions. Using the Star Wars API, see if you can create assertions to validate the following:

- Check that the server (this value is returned in a header) is nginx.
- Check that the response time for this call is less than 500 milliseconds.
- Check that Luke appears in 4 films.

You can check out my solutions to these in the GitHub repository for this book, but make sure you first try to figure them out on your own. Now that you have a good grasp of how to use assertions in a request, we should look at how to share them across multiple tests using folders and collections.

## Creating folder and collection tests

I have shown you how to create tests that can verify that a request is correct. Some assertions, though, might be the same across multiple tests. For example, you might have a set of positive tests that should all return a status code of 200. You can create an assertion in each of the requests that you set up, but there is an easier way to do it. In Postman, you can add test assertions to folders and collections.

If you have several requests in a folder and you add a test assertion to the folder, that assertion will run after each request in the folder has been completed. Let's look at an example with the following steps:

1. Add a folder to the **SWAPI** collection called Luke.
2. Drag the **Get First Person** request that you were working with earlier into that folder and then add another request to that folder called Luke's home world.
3. Set the URL of the request to `{{base_url}}/planets/1`.

4. Now, edit the folder and on the **Scripts** tab go to the **Post-response** section, and add a test to check that Luke and his home world both appear in film 1:

```
pm.test("Check that they are in film 1", function () {  
    var jsonData = pm.response.json();  
    pm.expect(jsonData.films).to.contain("https://swapi.dev/api/  
films/1/");  
});
```

5. **Save** the changes to the folder.
6. Go to the **Get First Person** and **Luke's home world** requests, and for each of them send the request.

You will notice in each case that the **Test Results** area shows that Postman has run the check that was defined in the folder. You can set up similar checks at the collection level as well. Any tests that you add to the collection will run any time a request in the collection is sent. Setting up tests in the collection works the same as doing it in a folder, so I will leave it up to you to play around with that.

Being able to share tests with multiple requests is helpful, but you may also need to clean up after yourself sometimes. You might have data or variables that you don't want to persist after a request has been completed, so how do you get rid of these?

## Cleaning up after tests

In many testing frameworks, you will have the ability to run teardown scripts. These are essentially scripts that let you clean up any variables or values that you don't want to persist once your test is complete. Postman does not have an explicit section where you can do something like this. However, if you have stuff that needs to be cleaned up after a test, you can do so right in the **Post-response** section.

One thing to be aware of here is the order of executions. Where should you put your cleanup scripts? Should you put them in the **Post-response** section for a request, or in a folder or collection? Scripts created at the folder or collection level will run after each request, but in what order should they be run?

After a request is completed, Postman will first execute the test scripts from the collection, then the script from the folder, and finally, the scripts from the request. This means that any variables created in a request test script can't be cleaned up in a collection or folder. In general, the best practice would be to clean up any variables in the same place where they are created.

So, if you make a temporary variable in a collection that you want to remove when you are done, do so in the collection where you made it. I think this is the easiest way to manage test data cleanup, but there may be times when you need to pass a variable around and so need to clean it up elsewhere. When doing so, knowing the execution order will help you avoid errors.

So far in this chapter, I have talked a lot about running tests after a request is sent. You have seen how to use the snippets in Postman to help you get started with creating tests. You've seen how to check various parts of a response, ranging from the body to the header, and you've also seen how to use the many built-in assertions that Postman provides. Now it is time to turn your attention to what you can do before a request has even been sent.

## Setting up pre-request scripts

Pre-request scripts work in much the same way that tests do. In this section, I will show you how to use them to set and get variables so that you can share data between tests. I will also show you how to build a request workflow where you can chain multiple tests together so that you can check more complex workflows. All these things are great on their own, but they do beg the question of how we can effectively run these tests, so this section will also cover how to run your tests in the collection runner.

The first thing we need to cover, though, is how to get started with pre-request scripts. These scripts use JavaScript to send commands just like the response assertions but, as the name implies, they are run before the request is sent rather than after. Now, why would you want to do that?

I have used pre-request scripts in a couple of different ways. I have had times when I wanted to test something in an API that required sending multiple API calls. In order to check the things that I wanted to check, I needed to be able to pass data that came back from one call into the next call. I could do this by assigning values to variables in the **Post-response** section of the first request and then reading those variables' values in the next test. However, there were times when I wanted to take one of the variable values and modify it slightly (for example, add one to it) before using it. In that case, I would use a pre-request script to do that.

Another example of a time I have used pre-request scripts is to test with some random data. I could generate a random number or string in a pre-request script and then use that value in the test itself. There are other ways that pre-request scripts can be useful as well, but one thing that you will often want to do with pre-request scripts is read or set environment variables.

## Using variables in pre-request scripts

In *Chapter 4, Considerations for Good API Test Automation*, I explained how variables work in Postman and the different scopes that they can have. In that chapter, I showed how to create and use variables in the user interface. Postman also lets you use variables in scripts. You can use them in Post-reponse scripts as well as pre-request scripts, but for now, I will focus on using them in the **Pre-request** section of the **Script** tab. The best way to learn how to do this is with a couple of examples. For the first one, create a new request in the **SWAPI** collection called **Get a Person**, and then use the following steps to create a pre-request script:

1. Set the URL of the request to `{{base_url}}/people/{{person_id}}`.

The `base_url` variable should already be defined from previous examples, but the `person_id` variable will not yet be defined.

2. Go to the **Pre-request** section on the **Script** tab for the request.
3. In the **Snippets** section, click on the **Set an environment variable** snippet. This will add the following code to the pre-request script:

```
pm.environment.set("variable_key", "variable_value");
```

4. In the set command, change `variable_key` to `person_id` and change `variable_value` to `1` so that you now have the following code:

```
pm.environment.set("person_id", 1);
```

Now, if you send the command, the pre-request script will first set the `person_id` variable to have a value of `1` and then, when the request is sent, it can use that variable to set the URL correctly.

This example was a simple and somewhat contrived one. I did this in order to help you understand how this works, but there is a lot more that you can do with this. For the next example, we'll look at how to pass data between tests.

## Passing data between tests

In order to pass data between requests, you will need multiple requests. You can use the request from the previous example for one of the requests, and create a new request called **Get Homeworld** in the **Star Wars** collection for the other one. Once you've done that, use the following steps to set up that request and pass data between the two requests:

1. Set the URL of that request to be `{{base_url}}/planets/1`.



2. On the **Post-response** section of that request, add the following code:

```
var jsonData = pm.response.json();
var planetResidents = jsonData.residents;
pm.collectionVariables.set("residentList", planetResidents);
```

This will get the list of URLs that represent the people who live on this planet out of the response from the first request and add it to a variable called `residentList`. Note that in this case, we are saving the variable into the `collectionVariables` scope. That is so that it will be available to all requests in the collection.

3. Send the **Get Homeworld** request in order to create that variable.

Now that you have a variable with a list of URLs, let's see if you can use it in the **Get a Person** request to get information about one of the residents of that planet.

4. On the **Pre-request** section of the **Get a Person** request, get rid of the previous code and instead add the following code:

```
var residentList = pm.collectionVariables.get('residentList');
var randomResident = residentList[Math.floor(Math.random() *
residentList.length)];
pm.environment.set("random_resident", randomResident);
```

5. Set the URL of the request to `{{random_resident}}`

Some of this code does things that you might not understand, so let's walk through this one line at a time and explain what is going on:

```
var residentList = pm.collectionVariables.get('residentList');
```

This first line is just getting the data in the `residentList` collection variable and storing that data in the `residentList` local variable:

```
var randomResident = residentList[Math.floor(Math.random() * residentList.
length)];
```

You only need to use one of the URLs in the `residentList` array, and so this next line picks a random item out of that list. It might seem intimidating to think of writing this kind of code, but if you are stuck on something like this, you can use an internet search to help you out.

For example, in this case, you might do an internet search for something like *how to get a random item out of a list in JavaScript*. Almost always, within the first couple of results, you will see an example that you can use. In this case, `Math.random()` is a JavaScript function that gives you a random number between 0 and 1. I then multiply it by the length of the list so that the number will be between 0 and the length of the list and then use `Math.floor` to round it down to an integer. That integer is then used as an index in the list to get the value at that index:

```
pm.environment.set("random_resident ", randomResident);
```

The final line in the script is just doing what we were doing in the previous example, but this time, instead of setting the value of `person_id` to 1, we are assigning the `randomResident` variable that we made to the `random_resident` environment variable that we are using in the URL. You should now be able to send the request and get back data about one of the inhabitants of Tatooine.

I don't know about you, but I think that is pretty cool! At this point, you could create a test for the random person call that checks that their home world is planet 1. I won't walk through the details of how to do that, but I encourage you to try it out for yourself, run the test a few times, and prove to yourself that this kind of powerful validation works! Now that you know how to pass data between tests, let's look at how you can leverage that to build workflows in Postman.

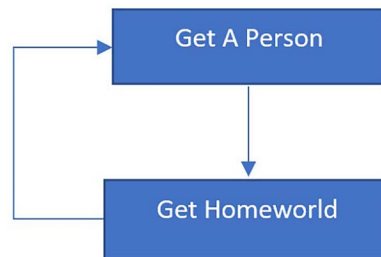
## Building request workflows

In this next section of the chapter, I will show you how to use environments to help manage data that is shared between tests, but there are also a few other built-in methods that can help you manage tests that are tied together like the example we just set up. In this case, the second request depends on the first request having been run. By default, Postman will run them in the order they appear in the collection, but what if someone moves them around? In order to explicitly tie them together, we can use the `setNextRequest` method to let Postman know which request it should run next. You could do this by adding a line like this to either the post-response or pre-request scripts to tell Postman that the next request to run is the `Get a Person` request:

```
postman.setNextRequest("Get a Person");
```

It is important to note that you still need to have the first request before the other one in the collection. The `setNextRequest` method will skip ahead to the next test (which means that any tests between them will not be executed).

If you had your tests in reverse order (that is, first **Get a Person** and then **Get Homeworld**), the execution order would look like this:



*Figure 6.7: Request execution loop*

First, the **Get a Person** request would run and then the **Get Homeworld** request would run. The **Get Homeworld** request would then call the **Get a Person** request (using the `setNextRequest` method). This would bring execution back to the top and so Postman would start running all the tests again. As you can see, this would lead to an infinite loop, so you want to be careful when using this method.

There are specific cases where using this makes sense, but in general, I would suggest that you do not use `setNextRequest` to control the flow between requests. You are better off designing your test collections well so that they run as you would want. However, there is one situation where I think looping can be helpful, and this method can help with that.

## Looping over the current request

You certainly don't want to set up any infinite loops, but sometimes you might want to use the same request multiple times with different values. The `setNextRequest` method can be used to help you do this. Go to the **Get a Person** request, and let's see if you can get it to run once for each person in `residentList`.

First of all, you don't need to pull a random value out of the list, so you can comment out that line by putting two slashes (`//`) in front of it, or you can just delete the line altogether. Instead of pulling out a random value, we want to run this test once for each item in the list. In order to know which instance of the loop you are on, you will need a counter. The problem is you can't just add a variable at the top of the script and initialize it to `0`, since then the counter would be at zero every time the request is run. You will need to have a variable that is not a part of the test. I will go over environments in more detail later in this chapter, but for now, just add this line to the script:

```
var currentCount = pm.collectionVariables.get("counter");
```

This counter needs to be initialized to start at 0. You can do this by going to the `Get Homeworld` request, and in the **Post-response** section, setting the environment variable to 0:

```
pm.collectionVariables.set("counter", 0);
```

Now go back to the `Get a Person` request and set it up to use that counter to access the items in the list:

```
var resident = residentList[currentCount];
```

This will give you the link to the index that `currentCount` is at. The counter starts at 0, so the first time this test is run, it will give you the first item in the list (don't forget that list indexes start at 0). Note that since you are no longer getting a random variable, you are now saving the value into a variable called `resident`. You will need to update the setting of the `resident` environment variable to reflect that:

```
pm.collectionVariables.set("resident", resident);
```

You will also need to update the URL to use the variable `{{resident}}` as the URL. You can now access the `resident` at the current counter value. However, we need that counter to change so that the next time we run this test, it gets the item at index 1. You can do this by setting the counter to equal one more than it does right now. Add this code to the bottom of the script to do that:

```
pm.collectionVariables.set("counter", currentCount+1);
```

You can create the loop by adding this code to the next line in the script:

```
postman.setNextRequest("Get a Person");
```

This will tell Postman that the next request you want to run is the `Get a Person` test, which of course is the current test. Postman will then run this request again, but since you have added 1 to the counter, it will now use the list item at index 1 instead of index 0. The test will continue to loop over and over, but there is still one more problem. When will the test stop looping? You need an **exit condition** so that you don't end up trying to run more loops than there are items in the list. In this case, the exit condition will be when the counter has accessed the last item in the list. You can create this condition with an `if` statement that checks whether `currentCount` is less than the length of the list:

```
if (currentCount < residentList.length-1) {
```

Note that we are subtracting one from the length of the list so that we don't run the request again once we've hit the end of the list. This if statement should be placed right after you set the resident environment variable and you should put everything else in the test inside that if statement. At this point, you should have the script ready to go and it should look like this:

```
var residentList = pm.collectionVariables.get('residentList');
var currentCount = pm.collectionVariables.get("counter");
var resident = residentList[currentCount];
pm.collectionVariables.set("resident", resident);
if (currentCount < residentList.length-1) {
    pm.collectionVariables.set("counter", currentCount+1);
    postman.setNextRequest("Get a Person");
}
```

Now that you have this script ready to go, let's look at how you would actually run it. The Postman collection runner allows you to run all the requests in a collection.

## Running requests in the collection runner

Before running the tests in the collection runner, make sure that you have saved the changes you made in the last section, and remember to ensure that the Get Homeworld request is first in the list. You can drag and drop the requests in the navigation panel to reorder them. Once you have the requests in the right order, you can open the collection runner. There are a few places in Postman to open it, but since you know that you want to run the tests in the **Star Wars API** collection, you can start it from there by following these steps:

1. Click on the collection in the navigation panel.
2. On the resulting page, there is a **Run** button near the top right. Click on that to open the collection runner:

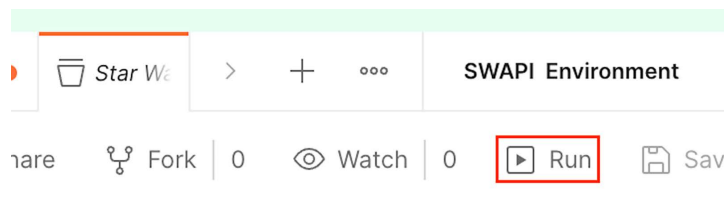


Figure 6.8: Run collection button

This will show you the order the requests will be run in, along with all the requests that will be run.

3. Deselect all the requests except the **Get Homeworld** and **Get a Person** requests, and make sure those two requests are in the correct order. Your collection runner should look similar to this:

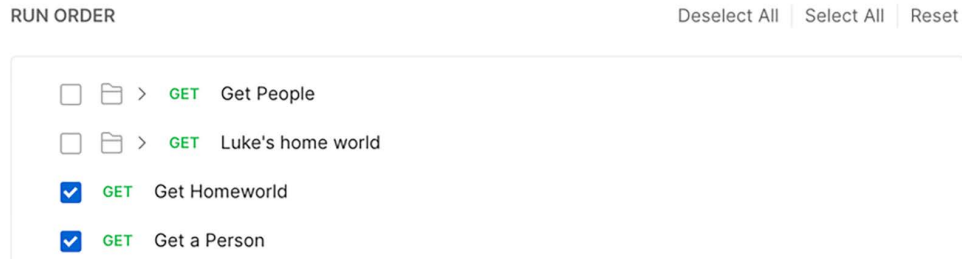


Figure 6.9: Collection runner setup

4. Once everything is set up correctly, click on the Run Star Wars API - Chapter 6 button to run the collection.

You will see that it runs the **Get Homeworld** request and then it runs the **Get a Person** request once for each person in the list.

There are some additional things to talk about on the collection runner, and I will go through some of the options in more detail in *Chapter 7, Data-Driven Testing*, but for now, you can see that you can use the collection runner to easily run multiple tests at once.

#### NOTE:



Before running requests in the collection runner, make sure you have saved any changes made to those requests. The collection runner uses the last saved version for each request when running. If you have made recent changes to a request and have not saved them, they will not be reflected in the collection runner. This can be confusing and has puzzled me in the past, so try to get in the habit of always saving requests before opening the collection runner.

Passing data between tests like this can enable some cool and powerful workflows, but there are some things to be careful of with this as well. One of the biggest issues you might run into with these kinds of workflows is challenges with maintaining the tests.

It can be a bit of work to track where a variable is being set or modified. In this example, I used the `collectionVariables` scope to store the variable, but you can also store variables in environments, which can make them a bit easier to manage.

## Using environments in Postman

Postman environments are a place where you can create and manage variables. You can set up multiple environments for different purposes. Environments are very helpful when you need to share data between requests, and so in this section, I will show you how to use them to manage your variables. You can also manipulate variables in the environment, and I'll show you how to do that as well so that you can also use Postman environments to help you explore or debug issues. In the first place, let's look at how to manage variables using environments.


### Managing environment variables

Creating an environment is easy. Follow these steps to create one:

1. Click on the **New** button and then select the **Environment** option.

You can give your environment a name – in this case, just call it something like **SWAPI Env**. You can then start creating variables in the environment. Let's use this environment to manage the `resident` variable that you are currently setting in the pre-request script of the `Get a Person` request.

2. Type `resident` in the first field of the **VARIABLE** column and set the **INITIAL VALUE** to `https://swapi.dev/api/people/1/`. The following figure shows what that would look like:

SWAPI Env  




	VARIABLE	TYPE 	INITIAL VALUE 
<input checked="" type="checkbox"/>	resident	default 	https://swapi.dev/api/people/1/
	Add a new variable		

Figure 6.10: Create an environment variable

3. Save this environment and then close the tab.
4. As shown in the following figure, in the upper right-hand corner of the app, you will see a dropdown. Click on this dropdown and select **SWAPI Env** from the list:

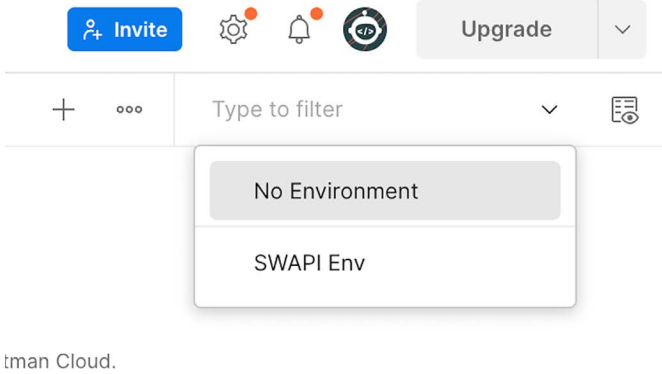


Figure 6.11: Select the environment

5. You will now need to switch where the variables in your requests are saving their data. In the **Post-response** section of the **Get Homeworld** request, change where the counter is getting set from `collectionVariables` to `environment`. Change the setting of the `residentList` variable in the same way:

```
pm.environment.set("residentList", planetResidents);
pm.environment.set("counter", 0);
```

6. Similarly, in the **Pre-request** section of the **Get a Person** request, update the getting and setting of the counter variable, the getting of the `residentList` variable, and the setting of the resident variable to happen in the environment instead of `collectionVariables`.
7. Now that the environment is active, send the **Get a Person** request.

This request will set the value of the resident variable to the current value of the resident URL.

8. Click on the **Environment quick look** icon to see what value it has been set to and you should see a panel pop up that looks like this:

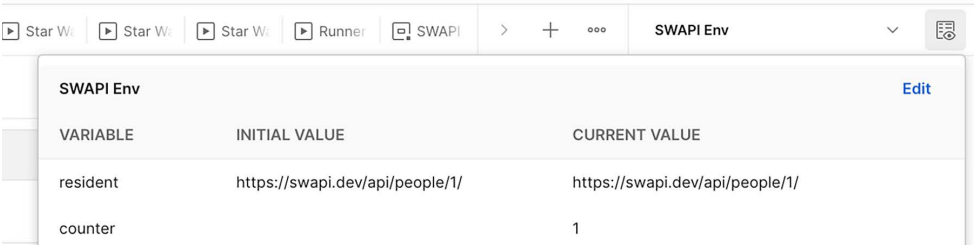


Figure 6.12: Environment quick look



If you send the request again and look at the environment variables, you will see that the current values are updated. You now have one easy-to-access spot where you can check on what value is being used for your variable.

One word of caution that I should mention again is to be careful about what data you put into the variables in an environment. If you are storing any variables with sensitive data, be sure to set their type to **secret**. Also, since environments can be shared, be careful about who you share them with.

## Summary

Sending API requests allows you to inspect the responses and check that the API call is working the way you want it to. This is a manual process, though, and often you will need to create tests that can be run over and over again to check for product regressions. In order to do this, you need to be able to add checks to a request. This chapter has given you the ability to do that. You have learned how to add assertions that check various aspects of an API response. You have also learned about the various built-in assertions that Postman provides to help you with this.

Sometimes you need to get some input data into a specific format in order to be able to check the things that you want to check. This chapter has also helped you learn how to set up scripts that can run before a request so that you can have everything in the necessary state. I also showed you how to run a request multiple times so that you can check many things in one request. This chapter also covered how to create request workflows so that you are able to check things that require multiple API requests.

In addition to being able to automatically check that API calls are working correctly, you need to be able to easily run those tests. In *Chapter 9, Running API Tests in CI with Newman*, I will show you more advanced ways to do this, but in this chapter, I introduced you to the collection runner, which allows you to run multiple tests. The chapter also taught you how to use environments to manage the variables and data that you might create in your testing scripts.

If you worked through the exercises and followed along with the material in this chapter, you should have a good foundation in place for getting started with creating automated tests. In the next chapter, I'll talk about some more advanced things that you can do with automated tests in Postman. We will be working through data-driven testing and how to set it up and use it in Postman.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.com/invite/nEN6EBYPq9>





# 7

## Data-Driven Testing

There are several testing techniques that can be used to improve the efficiency of test automation. One of those techniques is called **data-driven testing**. Test automation is a powerful way to speed up your testing ability, and the data-driven testing techniques explored here will help you speed this up even more.

Test automation allows you to try out many different things without getting bored. In fact, one of the heuristics I use for determining whether I should automate something is to ask myself whether what I am doing is boring. If I am working on boring and repetitive work, it's a good indicator that I might be working on something that should be automated.

Sometimes, however, creating test automation itself can get boring. You may want to create tests for several inputs to a request that are all quite similar. Creating good test cases can involve trying out many different inputs to the system. Automation is good at checking many things over and over, but it can take a lot of work to create separate requests for each of those inputs. So rather than doing that and duplicating a lot of work, you can use data-driven testing to increase the efficiency of your automated tests.

This chapter will teach you everything you need to know to use this powerful testing technique in Postman. By the end of this chapter, you will understand how to create powerful and scalable data-driven tests, use **Equivalence Class Partitioning (ECP)** to create good input for data-driven tests, define useful outputs for test comparisons, set up data-driven tests in Postman, create tests in Postman that can compare response results to data outputs in a file, and set up and run data-driven tests in the collection runner.

We will cover the following topics in this chapter:

- Defining data-driven testing
- Creating a data-driven test in Postman
- Challenge – data-driven testing with multiple APIs

## Technical requirements

The code used in this chapter can be found at <https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/master/Chapter07>.

## Defining data-driven testing

Data-driven testing is not unique to API tests. It can be used in other types of test automation as well. For example, you could set up UI tests or even unit tests to use data-driven testing techniques. In this section, I will explain what data-driven testing is. I will also teach you some principles that you can use to create good inputs for data-driven tests. In addition, this section will cover how to set up good test comparisons and the outputs for data-driven tests. But what exactly is data-driven testing?

Essentially, this technique involves creating a table of inputs that map to expected outputs. You would then run those inputs through the system under test and check whether the outputs of the tests match the outputs in the table. You may also hear this test technique described as **table-driven testing** or **parameterized testing**.

That is all a bit abstract, so let me try to explain it with an example to help you understand what this means. Let's imagine that you have a list of products in a catalog. You know the ID of each of the products, and you also know how much they are supposed to cost, and you want to verify that each product in the system is listed at the correct price. You could create a table in a .csv file that looks something like this:

ProductID	ExpectedPrice
123456	\$15.49
234561	\$12.99
876543	\$9.28
....	....

Figure 7.1: Product pricing table

The product IDs in this table give you a set of inputs, and the expected prices give you a set of outputs. Data-driven testing is merely the process of feeding those inputs into the test and checking that you get the expected outputs. In diagram form, it would look something like this:

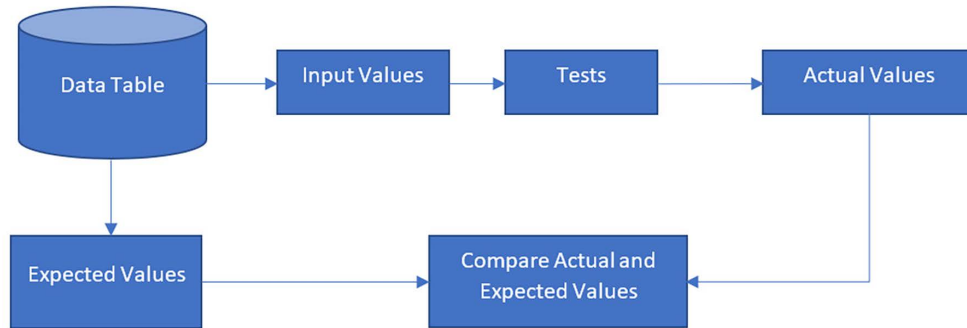


Figure 7.2: Data-driven testing workflow

In *Chapter 6, Creating Test Validation Scripts*, we saw how to create a test that called itself multiple times so that we could loop over a set of inputs. Data-driven testing is similar. Each input in the data table will be run through the test, and the result will be compared to the corresponding expected value in the table. The power of data-driven testing is that you can create one test and then pass many different values, thus leveraging that one test to check many things.

One thing to note about *Figure 7.2* is that the expected values do not need to be defined in the data table. For example, if you had many inputs that all have the same expected value, you could just hardcode that value into the test and then use the data table for inputs only. An example of this would be running a set of tests that iterates over a list of product IDs and checks that they all have a price greater than 0. In fact, this is quite a common way to do data-driven testing and is the default method that Postman supports. However, there are also times when you need to test with a different output for each input. Don't worry; in this chapter, I will also show you how to run data-driven tests where the expected values are defined in the data table.

Data-driven testing is a powerful technique, but as with most powerful things, it can be abused. Before deciding to use data-driven testing, there are a few questions you should ask yourself. First, you need to figure out whether there is a need for multiple inputs to a test. For example, if you were testing an endpoint that returns a given resource every time, there isn't much point in trying different inputs. Another case where you might not need data-driven testing is if the inputs are all very similar to each other. There is a bit of complexity involved in this, so let's dig into how to set up data-driven inputs.

## Setting up data-driven inputs

Say you are testing an endpoint that allows you to add two numbers together. All you need to do is call `/add` and include the two numbers in the body of the call. If you are trying to test the inputs sent to this, you could construct an infinite list of inputs. However, is there really a meaningful distinction to be made from a functional standpoint from passing the numbers 1 and 2 or the numbers 3 and 4? Unless there is a specific reason to check for a specific term, you don't want to create inputs that are essentially equivalent. Thinking through what inputs might be equivalent can be tricky, and this is where a testing technique known as **Equivalence Class Partitioning (ECP)** can be helpful. With ECP, you partition the inputs to your tests into various classes and then use only one input from each class in your test. The tricky part, of course, is deciding how to partition the inputs. However, thinking this through carefully can help you refine your inputs and reduce the number of iterations that you need to test. ECP is easiest to do if you deal with numerical or mathematical functions, but it can also be applied to text-based inputs. Let's use inputs to a password field as an example.

The broadest classes for a password field would be **Valid** and **Invalid**. Since a given username only has one valid option, we certainly can't break that class down any further, but what about the **Invalid** class? Are there any classes of invalid inputs that might be worth considering? If you think about where the boundaries of the field are, you can come up with two more classes: **Empty Password** and **Really Long Password**. You could also think about security concerns and come up with another class called **Injection Flaws**. I'm sure if you thought about it long enough, you could come up with a few more ways to partition the input data for this password field, but the point is that by breaking this down into four or five partitions, you can check one input for each of these partitions and know that the password field handles invalid inputs reasonably.

If you were to try and test all possible invalid inputs to the password, you would never finish. ECP gives you the ability to limit the inputs in a way that makes sense. In addition to defining the inputs for a data-driven test, you need to think about the outputs.

## Thinking about the outputs for data-driven tests

Carefully defined inputs can keep the number of iterations that you run a test for reasonable, but in order to set up useful tests, you need to also think about the outputs you use. You can have a well-designed set of inputs, but if you don't have good outputs, you aren't going to get any useful data out of the tests. There are two sets of outputs to consider. There are the outputs generated by the test itself, along with the outputs you define in your data table. Let's think about the ones defined in your data table first.

When you create a table for data-driven testing, you will need to know the answer to the question, “What should the expected result be of this input?” This means that the outputs in your data-driven testing table need to be easily computable. This might mean that they are trivially simple, as in the example of the password field where you know that one value should allow you to log in and all others should not. Or you might have an API that takes in a first name and a last name, returning a username in the form of `<first initial><last name>`. In this case, the expected result is also easily computable, since you can manually create a username that fits the pattern based on the inputs without too much difficulty.

However, there could be other inputs that are much harder to compute the correct outputs for. For example, there is an API (<https://github.com/aunyks/newton-api>) that allows you to do some complex math, such as finding the area under a curve. If you are not a mathematician, you might have trouble computing the answer for a given input. You could perhaps send a request and assume that the response you get back is correct, using that to check that things don’t change going forward, but if you test a brand-new API, that might be a difficult assumption to make.

Figuring out what the expected value for a response should be is not unique to data-driven testing, but it is something that you should consider when constructing the data table for these tests. Knowing the expected value is only half the problem, though. You need to be able to compare it to the actual value that you get from the test. Once again, this is not unique to data-driven testing, but there are some additional considerations that can come into play with data-driven testing. The comparison of an expected value to the actual result can be done using the assertions and techniques we covered in *Chapter 6, Creating Testing Validation Scripts*. However, with data-driven testing, you have multiple inputs, which means that you won’t get the exact same output for each row in the data table. When creating the comparisons in your tests, you need to be aware of this.

If you have a data-driven test that checks through a list of user IDs, you can’t set up an assertion that compares their email address to a static value. If you were only calling one user – say, someone called Jane Doe – you could check that the email address was `jane.doe@something.com`, but if you call the same tests with multiple users, you would need to check something more generic. In this case, for example, you could check that the email address field contains an `@` and a `.` or something like that instead.

When setting up data-driven tests, you need to think about both the inputs and outputs of your data table and the tests themselves. Now that you have a good grasp of how to effectively use data-driven testing, let’s take a look at how it works in Postman.



## Creating a data-driven test in Postman

Postman provides tools for running data-driven tests, but in order to use them, you will first need a test. This section will show you how to create an actual input file that you can use, and it will then teach you how to create the kind of test that you need for this in Postman. It will also show you how to get data from a file and use that data to compare it to the results of your request.

I will demonstrate all of this to you with a practical example. The example will use the API provided by JSONPlaceholder (<http://jsonplaceholder.typicode.com>). This is a sample API that you can use to get fake data. Before creating a data-driven test, follow these steps:

1. Add a collection called **JSON API**.
2. Add a request to the collection called **Users**.
3. Set the request URL to `http://jsonplaceholder.typicode.com/users/1` and send the request. You should get back a response with a bunch of data for a user.

Now that you have a working request, let's look at how to set up a data-driven test. Since a data-driven test takes in multiple inputs, you will need some kind of variable, or something that changes with each input. That is called a **parameter**, and for this example, I want you to parameterize the user ID in the URL. You can do that by turning it into a variable with the following steps:

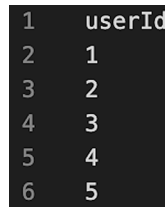
1. Replace the 1 in the URL with a variable called `{{userId}}`.
2. For ease of management, create an environment to save this variable in by clicking on the **New** button and choosing the **Environment** option.
3. Name the environment **JSON API Env**, create the `userId` variable with an initial value of 1, and then click the **Save** button to create the environment.
4. Choose **JSON API Env** from the environment dropdown at the top right of the Postman application.

Now, if you go back to the request you made and mouse over the `userId` parameter in the URL, you should see that it has a value of 1, and if you were to send the request again, you would get back the same response as you did previously. With the input parameter defined, it's time to look at how to create the data that you will use to drive the test.

## Creating the data input

Input files for data-driven tests are often created as `.csv` files. If you have access to Excel or another spreadsheet program, you can use that to make the file, but otherwise, you can easily create the data in any text editor.

For the purposes of this demonstration, I want you to create a file that will run for the first five users in the list. The first row of the file is the header and will have the name of the parameter that you define. This parameter name needs to match up with the name you use in the test, so in this case, `userId`. It is important that the name you use as the header for your column exactly matches the name of the variable you use in Postman. On the next line, put the first user ID as 1, and then on the next line put the user ID 2, and so on until 5. Once you are done, you should have a file that looks something like this:



1	userId
2	1
3	2
4	3
5	4
6	5

Figure 7.3: Input data

Save this file, calling it something such as `userInputs.csv`. In this case, you only define inputs and not outputs. I will show you how you can set up outputs later in this section. For now, let's look at how to run a simple data-driven test. In fact, for this first run, let's keep it so simple that we won't even add an assertion to the request. Let's just run the request once for each input in the file, by following these steps:

1. Before running tests, it is important to ensure that you have saved any changes in your request. The collection runner will use the latest saved version of any requests that you run, so go to the **Users** request that you created and ensure that you have saved all changes.
2. Open the collection runner by going to the JSON API collection and clicking the **Run** button at the top right of the **collection** tab.

This will open the collection runner for you. Ensure that the **Users** request is selected in the **run order** panel.

3. Click on the **Select File** button under the **Data** header, browse to where you saved the `userInputs.csv` file, and open it.

When you do this, you will notice that Postman sets a few things for you automatically. It detects that you have five rows of data in the file, so it sets the **Iterations** field to 5. It also detects that the file type is `text/csv` and chooses that option for you. You can also preview the data that Postman will use for each iteration by clicking on the **Preview** button.

4. Click on the **Run JSON API** button to run the request.

The collection runner will call the request once for each line in the file, passing each respective `userId` value to the request. Of course, without any tests, this is a pretty boring test run, so let's look at adding a test to the request.

## Adding a test


In order to actually do something useful with the request, we are going to need at least one test so that we can verify that things work correctly. In order to do this, close the collection runner and go back to the request you created.

As I discussed in detail in *Chapter 6, Creating Test Validation Scripts*, you can create a test by going to the **Post-response** section of the **Scripts** tab of your request. For this example, set up a check that verifies that the email address field has an `@` symbol in it. You can use the `Response body: JSON value check` snippet to get started. You can then modify it to get the email address field, using `jsonData.email`, and check whether it contains the `@` symbol. Rename the test as well, and the final code for the test should look like this:

```
pm.test("Check for @ in email", function () {  
    var jsonData = pm.response.json();  
    pm.expect(jsonData.email).to.contain("@");  
});
```

Now, if you rerun the data-driven test in the collection runner (don't forget to save the request first!), you will see that it calls each request with different user IDs and then checks whether the email field on each request has an `@` in it. You should see something similar to the following:

### JSON API - Run results

 Ran today at 08:03:31 · [View all runs](#)

Source	Environment	Iterations
Runner	JSON API Env	5

All Tests Passed (5) Failed (0) Skipped (0)

#### Iteration 1

##### GET Users

<http://jsonplaceholder.typicode.com/users/1>

 PASS Check for @ in email

#### Iteration 2

##### GET Users

<http://jsonplaceholder.typicode.com/users/2>

 PASS Check for @ in email

Figure 7.4: Result of the test run

Now that we are actually checking something in the response, these tests are much more interesting, but what if we wanted to check something more specific? What if we wanted to check that certain user IDs match certain email addresses? We can't create an assertion in the test that checks for this because the email address will change as we run through each iteration of our dataset, so what can we do? Well, if we don't want to create a separate request for each of the users, we will need to define the expected email for each user ID in our file and then compare the emails from the API responses to those that we have defined in our file.

## Comparing responses to data from a file

Comparing data from an API response to data from a file requires changing things in a few different places. You will obviously need to add some data to the file itself, but you will also need to get that data into the test assertion that does the comparison between the actual and expected values. You can set this all up with the following steps:

1. Open the `userInputs.csv` file and add a new column to it.
2. Create a header called `expectedEmail`. This will be the variable name that you will use in the test.
3. Add the email address to each row that corresponds to the user ID for that row.

You may need to manually call the endpoint with each user ID in order to find out what the correct email address is for each row.

4. Modify one of the email addresses so that it is incorrect. It is always a good idea to check that your tests will actually fail if something goes wrong.

Once you have modified your file, it should look similar to what is shown in *Figure 7.5*. Make sure that the file is saved:

```
userId, expectedEmail
1, Sincere@april.biz
2, Shanna@melissa.tv
3, Nathan@yesenia.net
4, IncorrectEmail@kory.org
5, Lucio_Hettinger@annie.ca
```

*Figure 7.5: Data table with inputs and outputs*

With the file set up with output data, you now need to use that data in your test assertion.

5. Go to the **Tests** tab of the **Users** request in Postman. Right now, it is set up to expect the email of the response to contain an @. Instead, I want you to change it so that it checks whether the email is equal to `expectedEmail`. After making those changes, the code should look like this:

```
pm.expect(jsonData.email).to.eql(expectedEmail);
```

However, the problem is that the `expectedEmail` variable in this line is not yet defined. Test scripts can't directly use a variable in this way. Instead, we must extract the variable from the place where these kinds of variables are stored. There are a couple of different ways to do that in Postman. One way to do it is to use a special variable called `data` that Postman uses to store variables that get passed in from a file. With this variable, you can get the `expectedEmail` value using the following code:

```
var expectedEmail = data.expectedEmail;
```

Although this works, I think it could be confusing to people looking at this test in the future. Unless you know that the `data` variable contains a map to variables passed in by a file, you won't understand this line of code. Postman provides several other methods to interact with variables in scripts, and I think it would be better to use one of those instead if we could. Thankfully, there is a method we can use for this – the `replaceIn` method. You can use it in the following way to get `expectedEmail` into your test script:

```
var expectedEmail = pm.variables.replaceIn("{{expectedEmail}}");
```

Now that you have the expected email variable from the file, rename the test to something such as `Validate Email Address From File` to more accurately reflect what it does.

If you save all changes to the request and then open the collection runner, you can select the modified file and run the JSON API collection. Once you do so, you should see results like those in the following figure, where the second-last result fails, verifying that the check you have created indeed works:

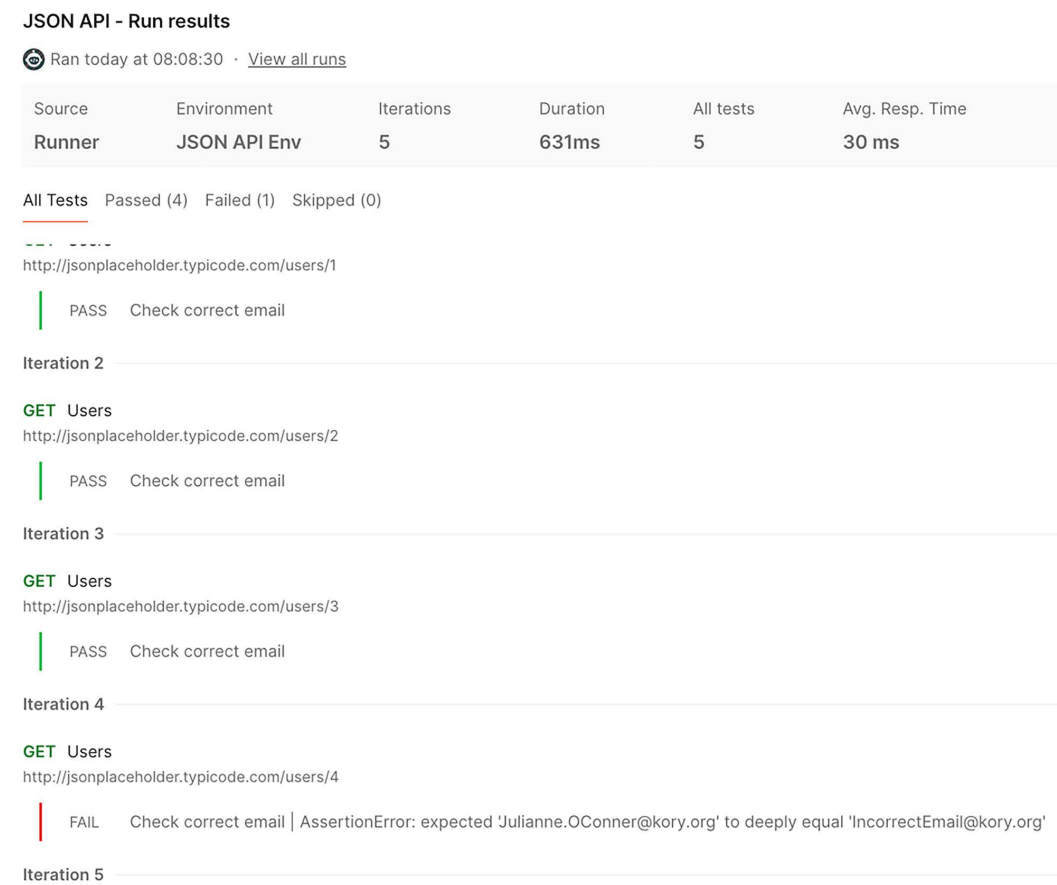


Figure 7.6: Test result of a data-driven test run

I only have space in this book to show you this one example of using data-driven testing functionality in Postman, but I hope that it has illustrated the power of this technique. I really want to make sure that you understand how to use this technique, so let’s turn to some hands-on practice.

## Challenge – data-driven testing with multiple APIs

I don't have the space to work through another full example, but I would like you to practice this a little more. I'm going to give you a challenge that I think will be, well, challenging. In this section, I will get you started with the challenge and give you a few hints to help you successfully solve it. In order to solve this challenge, you will need to use everything you have learned in this chapter about data-driven testing, but you will also need to use some of the information covered in *Chapter 6, Creating Test Validation Scripts*. In that chapter, I explained how to create tests that use multiple requests, and you are going to need that information to solve this challenge.

### Challenge setup

For this challenge, I want you to use the Postman Echo API. This sample API has a number of different endpoints that you can use. In this case, you will use the **Time Addition** (<https://postman-echo.com/time/add>) and **Time Subtraction** (<https://postman-echo.com/time/subtract>) endpoints. Each of the endpoints will take in a timestamp as a query parameter and then add or subtract the given amount of time from that timestamp. You can find the documentation on how to use them here: <https://docs.postman-echo.com/#4cef08e8-75d3-2a31-e703-115cf976e75e>.

Your job is to test these two APIs using data-driven testing techniques. You will need to set up a file that has the necessary inputs and then verify that you get the correct outputs. There are two ways you could check the correct outputs. You could just manually figure out the answers. If you apply a timestamp and you add 10 days, it's pretty easy to manually figure out the answer. However, that could be a bit tedious to create for multiple input values, so instead, I want you to use a different method to validate the answers.

I want you to start with the `/time/add` endpoint. Pass it timestamps and different units of time to add to the timestamp. Then, use the `/time/subtract/` endpoint to subtract the same amount of time from the result of the first request, and verify that you get back the initial timestamp.

### Challenge hints

This challenge is going to require reading data from a file to drive the test, as well as passing data between tests. This could get complex, so here are a few hints to help you as you try to figure this out:

- The query parameter that constructs the datetime instance can be sent as `timestamp`, `locale`, `format`, or `strict`. For the purpose of this challenge, don't worry about testing all those variations. You can stick with the `format` parameter to specify the datetime instance.



- The add and subtract endpoints return the dates in a fully expanded format (something like `Mon Oct 12 2020 00:00:00 GMT+0000`). Since you are going to be comparing the input dates to the dates that are output by the `/time/subtract` endpoint, you will want to create your input dates in a similar way.
- Don't forget about the order that you need to run requests in when passing data between them.
- Manually send requests to the two endpoints to make sure you understand the format of the responses that they give.

With these hints, along with what you have learned in this chapter and the previous one, you should be able to come up with a way to complete this challenge. I would recommend that you give yourself at least 30–60 minutes to work through this exercise. Experiment with different things, and try to get to the point where you can run the two requests in the collection runner with several different input values that you have defined in a file `DataDrivenTesting.postman_collection.json`.

## Summary

Data-driven testing allows you to do a lot more with your tests. It allows you to create test cases in external sources and then use those test cases to drive your tests. It also lets you leverage the power of automation much more effectively. With just a few tests, you can try out many different inputs and variations and learn a lot about the system under test. Data-driven testing is one of those testing techniques that you can really only use with automation, and it is techniques like this that make test automation so useful.

In this chapter, I have given you a good grounding in understanding what data-driven testing is and when it makes sense to use it. I have shown you how to set up data inputs and how to think about and use outputs in data-driven testing. You have also learned how to apply this all in Postman and seen how to create data inputs that Postman can use. In addition, you have seen how to add tests that can get variables that are defined in an external file, using them to dynamically compare against various test results.

You have seen how to set up and run these kinds of tests with the collection runner. In the next chapter, we will look at the tools Postman has to run workflow tests where you need to run a sequence of APIs in a certain order.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.com/invite/nEN6EBYPq9>





# 8

## Workflow Testing

Testers need to think of the big picture. Testing isn't just about the details of what inputs to send and what outputs to expect. It is about checking for risks to the client or the business. API testing is no different. We need to check on the kinds of things that matter to the users. Often, with APIs, this involves some kind of workflow where multiple endpoints are called, sometimes in succession.

Workflow testing is often more complex than endpoint testing. It is worth spending some time digging into it and so, in this chapter, we will look at how to approach this kind of testing.

We will cover the following topics in this chapter:

- Different types of workflow tests
- Workflow testing with the Flows feature in Postman
- Advice for creating workflow tests

### Different types of workflow tests

We already looked at how Postman runs requests in a couple of the previous chapters. If you create a collection with a series of requests in it and then run that collection, the requests will be run in the order in which they are listed in the collection. As mentioned in *Chapter 6, Creating Test Validation Scripts*, you can override the order in which tests are run by using the `postman.setNextRequest` command. This is an example of creating a testing workflow, but there are also a few other ways to do this.

# Linear workflows

The simplest workflow is a linear workflow. What I mean by a linear workflow is just a set of requests that run in a given order but that have some shared thread that runs through them. The distinction between a linear workflow and a collection with a set of requests is that, in the linear workflow, the requests will have dependencies on each other. If you were to change the order in which the tests were run, things would break.

Think of adding items to a cart in an e-commerce application. In order to add items to a cart, you need to first have a cart to add them to. Let’s imagine that there is a separate API call to create the cart. You would need to call that endpoint before you could call the one to add items to the cart. If you ran the add items call first, things would break, and so this is an example of a linear workflow.

Let’s look at an example of setting up a linear workflow. In order to do this, we will use a simple to-do list app that I wrote. The easiest way to run this application is in Gitpod. This should have all the dependencies set up for you automatically. You can do that with the following steps:

- 1. First of all, you will need a GitHub account. If you don’t yet have one, you can sign up at <https://github.com>.
- 2. Once you have that account, navigate to <https://gitpod.io/#https://github.com/djwester/todo-list-testing> in your browser.
- 3. Click to continue with GitHub.
- 4. You can accept the default workspace settings and continue.
- 5. Once it has finished loading, go to the terminal, type in the command `make run-dev`, and hit *Enter*.
- 6. The service should start, and you should see a toast message with a few options. Click on the **Make Public** option.
- 7. Click on the **Ports** tab and you will see the public URL of the test site available to copy.

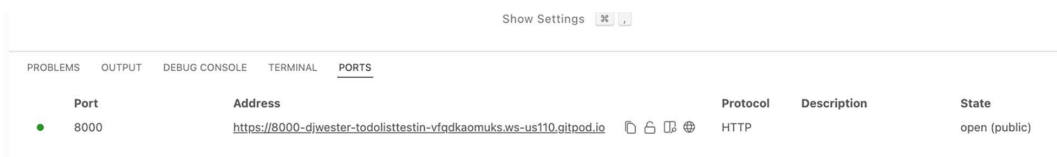


Figure 8.1: Finding the public URL

**NOTE:**

Gitpod will shut this site down after a few minutes of inactivity, so if you haven't been using it for a while and you get an error when making API calls, you may have to come back and repeat these steps to restart the site. Also, every time you restart the site, it will have a slightly different URL so don't forget to update the URL of any API calls pointing at this site.

The Gitpod-hosted version of this application should work well for the examples we will work through, but if you do want to set it up to run on your own computer, you can do so by following the directions in the GitHub repo for it at <https://github.com/djwester/todo-list-testing>.

Now let's turn to creating a linear workflow test:

1. Start by creating a collection in Postman called `Linear Workflow`.
2. We will want a `base_url` variable, so create an environment called `ToDo Env` and create a variable in it called `base_url`.
3. Set the value of it to the location of the to-do list app. If you used Gitpod to start this site, this will be the public URL for the site.
4. Make sure to save the environment and ensure that the environment is selected as the active environment.
5. Create a request in the collection and set **URL** to `{{base_url}}/tasks`. Name the request `Create ToDo`.
6. Set the method to **POST** and, on the **Body** tab, choose the **raw** option and set the type to **JSON**. Create the following JSON object in the body:

```
{
  "description": "Finish reading this book",
  "status": "Draft"
}
```

7. Save the request and then create a second request in the collection called `Edit ToDo`.

The URL of this request is going to require the ID of a task. Since this is a linear workflow, we are going to get this ID from the previous task that we just created. In order to do this, we are going to have to go back into the first request that we created and save the ID of it.

8. On the **Tests** tab of the **Create ToDo** request, add the following code to save the task ID:

```
var jsonData = pm.response.json();
```

```
var id = jsonData.id

pm.collectionVariables.set("task_id",id);
```

You should now be able to use the `task_id` variable in the edit request.

9. Set **URL** of the **Edit ToDo** request to `{{base_url}}/tasks/{{task_id}}` and set the method to PUT.
10. On the **Body** tab, choose the **raw** option and set the type to **JSON**. Create the following JSON object in the body:

```
{
  "description": "Finish reading this book",
  "status": "In Progress"
}
```

11. Create a third request called **Delete ToDo**.
12. Set **URL** of this request to `{{base_url}}/tasks/{{task_id}}` and set the method to DELETE.

You can now run this collection and have a linear workflow.

A reasonably complete API testing suite will usually include some linear workflows. These tests are necessary for checking that data can flow between endpoints that are meant to go together or that depend on each other to work correctly. Building this type of test is going to be an essential part of any API tester's toolbox.

## Business workflow

Another kind of workflow is something that I will call a business workflow. A workflow like this is interested in running through a business objective. It might require integration with more than one system, and it might require a certain setup to occur. You will often see these kinds of workflows automated in UI smoke tests using tools like Selenium. However, many applications now provide quite comprehensive API support, which makes it possible to automate these kinds of workflows at the API level.

An example of a business workflow could be a data processing system that first imports some data from an external source and then needs to wait for a little while as the data gets processed; then it can do some queries to test the data.

If this was being run through API calls, it might look a bit like this:

1. Call an API endpoint to trigger a data import into your application.
2. Poll an API endpoint that shows the status of the data import.
3. Once the data import has completed, call an API endpoint to start processing the data.
4. Poll an API endpoint that gives you the status of the data processing.
5. After the data processing completes, make some API calls to validate that the data has been correctly processed.

As you can imagine, a workflow like this can get a bit tricky to implement. You might need to put in a delay or have some kind of loop that keeps polling a status endpoint until it gets a certain response. You can build these kinds of things in Postman by creating requests in a collection and using scripting to make waits or loops or other things that you need. However, Postman also has a feature called Flows that gives you the ability to create some of these things a little more easily. Let's look at it in the next section.

## Workflow testing with the Flows feature in Postman

The Flows feature is a relatively new feature in Postman. You can create a flow by going to **New** and choosing the **Flows** option:

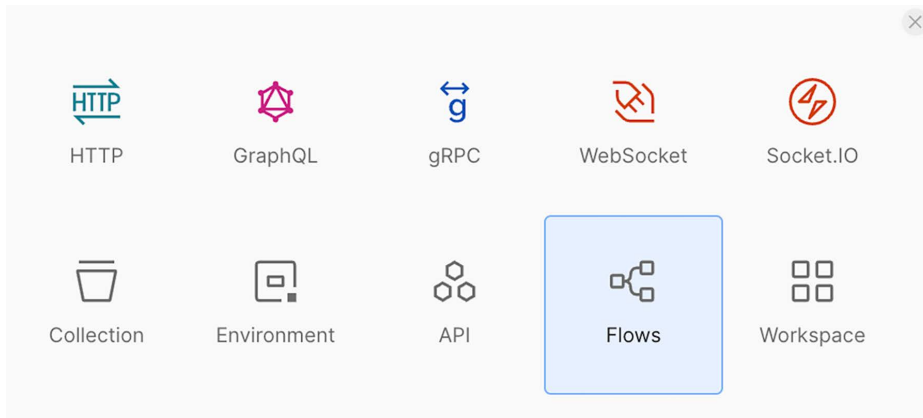


Figure 8.2: Creating a new flow



Usually, the first thing you will want to do with a flow is to add a block.

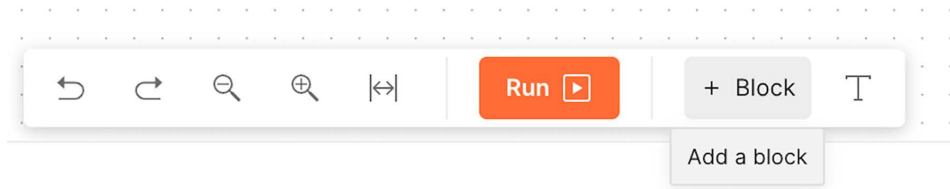


Figure 8.3: Adding a block to the flow

Once you've clicked **Add a block**, choose the **Send Request** option and then click anywhere on the canvas to place the block.

## Configuring a Send Request block

You can configure the **Send Request** block with the following steps:

1. Click on the **Select a request** field and start typing `todo` to search for the requests that you've previously made.
2. From the **Linear Workflow** collection, choose the **Create ToDo** request.
3. Click on the **Add environment** field and choose **ToDo Env**.

You will notice that it has identified that this request relies on the `base_url` variable. The value of this variable is already set by the environment that you've selected, but if you wanted, you could instead manually specify it. To do that, you would click on the **+ Click to add data blocks** option and choose the **String** option. You could then type in the value you want this variable to have.

Now that you have the block configured, let's look at how you can use it as part of a flow. You can see that beside the **Send** tag on the block, there is a little bubble. Click on this and drag over to the **Start** button to connect the two blocks.



You will also need a request to modify the status of a given task, so add a second request to the collection and call it `Update Status`:

1. Set the method of this request to **PUT**.
2. Set **URL** to `{{base_url}}/tasks/{{task_id}}`.
3. On the **Body** tab, choose the **raw** option and then choose **JSON** from the dropdown. Add the following body (don't forget to save once you are done):

```
{
  "status": "In Progress",
  "description": "{{description}}"
}
```

Now let's build our workflow with these requests:

1. Create a new flow.
2. Add a new send request attached to the **Start** block.
3. Select the **Get All Todos** request from the **Flow Requests** collection.
4. Make sure to set the environment to the **Todo Env** environment.

Now that we have this request, we need to get the response from it so that we can loop over the whole thing. We can do that with a **Select** block.

5. Click on the **Success** bubble at the edge of the Send Request block, drag it a little way, and then release the mouse button. When you do so, you should get a search bar. Search for **Select** and choose the **Select** block.
6. For the path, type in `body`. This should give you the body of the response, which will contain a list of todos. We want to loop over these items.
7. Once again, drag a connection off the end of the **select** block you just made, and when you release your mouse, put it into the search. Select the **For** block. This will loop over each of the items in the returned list.

At this point, your workflow should look something like this:

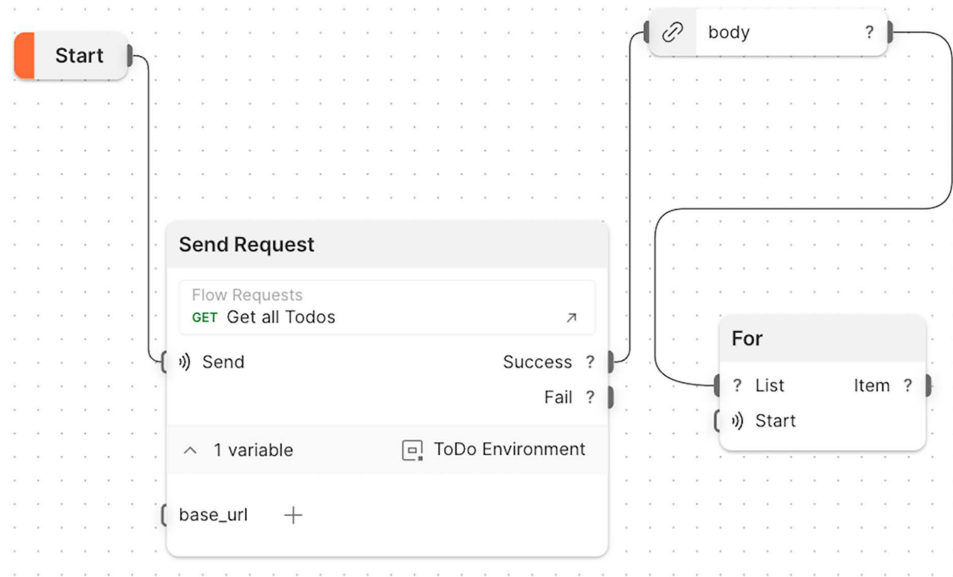


Figure 8.5: Start of workflow setup

For each of these todo items, we want to send an `Update Status` request, but we only want to do so if the status is **draft**, so the next thing we need to add is an **If** block.

8. Drag a connection off the end of the **For** block and create an **If** block. Set the path of that block to **status** in order to select the status of the current todo item.
9. You will also need to rename the variable. Double-click on `value1` and rename it to `status`.
10. You can now put in a check on the value of this variable. Type the following into the text field on this block:

```
$contains(status, "Draft")
```

When this statement evaluates to true, you will want to pass the data through this block so that you can update the status of the todo item. In order to do this, you will need to pass the data from the current iteration of the for loop to the **if** block.

11. Drag a connection from the **Item** bubble of the **For** block to the **Data** bubble of the **If** block.
12. Drag a new connection from the **TRUE** bubble of the **If** block and create another **Send Request** block.
13. Set up this **Send Request** block by choosing the **Update Status** request from the **Flow Requests** collection and ensuring that you have **ToDo Env** set.

You will notice that this block has several variables. The `base_url` variable will be automatically set by the environment that you selected, but you will need to get the other two variables from the **select** blocks that you just made.

14. Drag a second connection from the **THEN** bubble of the **If** block to the **task\_id** field on the **send** request. Set the path for this to `id`.
15. Drag a third connection from the **THEN** bubble of the **If** block and link it to the **description** field on the **Send** request. Set the path of this variable to `description`.

This flow should be working now, but there is one last block we will add.

16. Drag a connection from the **Success** bubble of the **Update Status Send Request** block, search for output, and create an **output** block. This block will display the results of the output.

Now it is time to run it! First, make sure that you have at least one task that is in the **Draft** state and then click on the **Run** button. You should see some dots moving along the flow to show you what it is executing and, in the end, you should see the **Output** block show you the output for each draft task that it updates to be in progress. The final flow should look something like *Figure 8.6*. Of course, your blocks might not all be in the exact same places as these, but you should have all of these blocks:

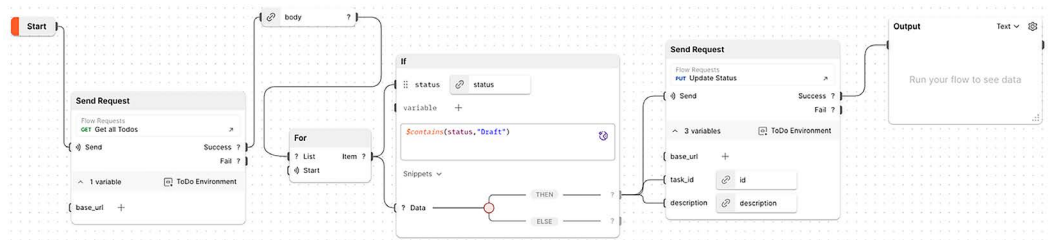


Figure 8.6: Completed flow

As you might imagine, you can do a lot more with flows in Postman. Even this flow could be fleshed out further. If you were to run this flow when there were no todo items in the Draft status, the **If** block would evaluate to `false` and the flow would go to **ELSE**. The workflow isn't currently doing anything with that data, but you could try adding some blocks from there to help you better understand how this works. If you are interested in learning more about flows, you can check out the Postman documentation for this. You can see some examples in there of how to integrate with apps like Slack so that you can automatically send reports when a flow completes.

Flows is a newer feature in Postman, so it might continue to change and evolve, but I think it's still worth checking out in its current form if you are currently trying to automate some more complex things.

## Advice for creating workflow tests

Now that you know how to design different workflows and some of the different ways that you can use Postman to automate them, we should spend a bit of time thinking about testing them. Basic testing principles apply, but there are also some additional things to think about with this.

Often, API testing will focus on checking that atomic actions (like sending a single request) work correctly, but workflow testing looks at how an API fits into a large ecosystem. Whatever type of workflow you are building and regardless of what part of Postman you use to build it, there are a few testing considerations to keep in mind.

## Checking complex things

Building the right checks into your tests is a skill that can take some time to master. The checks for an atomic API test can often be straightforward. You just need to check that it returns a **200** response and check for some basic data structure, for example. However, with more complex workflows, the checks can sometimes get more complex. When we are testing workflows, we are not just checking that the individual pieces of an API work; we are also checking the business logic that this API enables.

Let's think through an example. Imagine again that we are trying to test a business flow like the one outlined earlier, where data is imported into the application and then processed. We can use various API calls along the way to build this process, but what kind of things would we want to check with this?

Let's say the workflow is made up of the following API calls:

1. A POST call to `/import` that passes in a `.csv` file with hundreds of rows of data that immediately returns a job processing ID.
2. A GET call to `/importStatus/{job_processing_id}` that will return complete when the run is done.
3. A POST call to `/processData/{job_processing_id}` that will process the data into another form.
4. A GET call to `/businessData` that will return the requested business data. This data could include both the data that we just processed and some other data that was already there.

If you were just testing that final call to `/businessData` all on its own, it would be a simple test. You would have some data in the system, and you would call the endpoint and verify that it is there. But with this workflow, you need to check a bit deeper. You need to check that none of the data that you imported was lost along the way. You also need to check that it has been processed correctly so that it is transformed into the business data that you need. There are a couple of hints I have for building complex tests like this.

These are business-level tests, so the first piece of advice is to talk to the business. Talk to the project managers, business analysts, or whoever it is in your company that is responsible for figuring out the business workflows. Make sure you understand why this flow is there and what it is supposed to be doing so that you can ensure you are testing the right things. Also, if it is possible, talk to the customers. Who is going to be using this workflow and what are they going to use it for? The better you can understand the way customers are going to use the workflow, the better you will be able to design tests that validate it.

Another important thing to keep in mind when making these kinds of tests is that you should not have too many of them. These are costly tests on almost every level. They take a lot of time to make and to run. They also tend to be time-consuming to maintain. It is good to have one or two workflow-level tests that check that things are correct at the level that clients need, but keep it to just one or two.

## Checking things outside of Postman

As you design workflow tests, one of the things that will likely come up is that the API calls will only be one part of the workflow. There might be a final step that requires data to be in another system without a usable API or some processing of data that needs to occur in a third-party service. Whatever the case, there are times when you won't be able to keep the full workflow in Postman. What do you do in those circumstances?

Since this book is focused on Postman, I won't spend much time on this, but there are a few things that would be helpful to discuss now.



### NOTE:

We will also talk about this in a little more detail later in the book when we cover running Postman in CI builds.

The first point is that automation can be very helpful without being complete. If you are testing an important business workflow and you are trying to automate it, even if you end up only being able to automate the parts of it that have APIs, that can still be helpful. It can speed up and simplify those parts of the workflow even if you still have to do some manual steps before or after. Sometimes, we take an all-or-nothing attitude to our test automation. Don't forget that it is perfectly fine to start with partial automation and then build on it over time.

Another thing to consider with these kinds of workflows is to see if you can log a record of the data that will go to the next part of the workflow. As you move between systems, if you can make it so that you can restart a workflow from an intermediate point, it will help with debugging in the case of failure. If the workflow test fails, you can go back to the point where you logged the data and see if the problem had already occurred there. If it did, you know that the problem was at some earlier point in the workflow; if not, you know you need to look at the last part of the workflow. Logging is important in any big test, but when you have a natural seam in your test where you are passing data from one tool to the next, it makes sense to log data at that point.

If you want to fully automate a workflow that requires more than one tool, you will probably need to build a custom testing harness or, at the minimum, a custom build runner that would let you do this. This kind of thing goes beyond the scope of this book, but don't let that stop you from designing these types of tests. If a test is important and fits in with your strategy, run that test. Automate whatever you can, and get help if you need it, but never be afraid of making complex tests if you need them!

## Summary

In this chapter, we have discussed some of the different types of workflow tests that you might want to create. We have also looked at the Flows feature in Postman, which allows you to construct certain kinds of workflow tests that might otherwise be difficult to make. We have also talked about some of the testing considerations that you want to keep in mind with these kinds of tests.

Workflow testing is an important part of an effective testing strategy and UI testing isn't the only place you should be thinking about it. As more and more applications provide comprehensive APIs, it is worth looking into how much of your workflow testing can be done at the API level.

In the next chapter, we will look at the command-line tool Newman. We will see how to use it to run API tests at the command line and in CI.



## Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below to get a free eBook of your choice.



# 9

## Running API Tests in CI with Newman

The year was 2007, and I was hanging out with some friends who were techies. They always had the latest tech gear and in 2007, the latest tech gear was a new phone that Apple had put out. A phone that didn't have a physical keypad. A touchscreen phone! Can you imagine that? They were showing off their new iPhones and the rest of us were gathered around to check out this crazy new phone. I remember trying it out and struggling to figure out how to navigate it. I thought it was all going to be a fad and we'd be back to using our Blackberries, with their physical keyboards, soon.

Fast forward a few years and along with almost everyone else in the world, I had my own smartphone. The meteoric rise of handheld computing devices has changed a lot of things in our culture. These touchscreen devices are so intuitive and handy. However, there is one thing that most of us would rarely do on a mobile device, and that is to use its command prompt.

The command prompt (or terminal or shell as it is also called) had already become a less frequently used tool in computing due to better and better operating system user interfaces, but with the smartphone era, we are less likely than ever to use it. I think that is too bad. The humble little command prompt has a lot to offer.

In this chapter, we will learn all about Newman, which is a test runner that you can execute – you guessed it – from the command prompt. Newman allows you to run Postman collections from the command prompt and it enables a lot of powerful abilities for test automation. If you have become used to operating things through the great user interfaces that we have on our phones and computers and feel intimidated by the thought of running commands directly in the command prompt, don't worry – I will be sure to explain things as we go.

You will be able to learn everything you need to know as you go through the examples in this chapter. The focus of this chapter will be on learning how to use Newman, but there will also be a few other things that we will need to do with the command prompt. I will explain those things as we go.

The topics that we are going to cover in this chapter are as follows:

- Getting Newman set up
- Understanding Newman run options
- Reporting on tests in Newman
- Integrating Newman into CI/CD builds

## Technical requirements

The code that will be used in this chapter can be found at <https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/master/Chapter09>.

## Getting Newman set up

Newman is a tool built by the team behind Postman so that you can run Postman collections from the command line. It is also maintained by the Postman team and has feature parity with Postman, which means that you should be able to do anything in Newman that you can in Postman. However, Newman is not installed when you install Postman, so you need to install it separately. In this section, I will show you how to get started with Newman. I will go over how to install Newman, as well as how to use it at the command line. In order to do this, I will also show you how to install the Node.js JavaScript framework and how to use its package manager to install Newman and other helpful libraries.

## Installing Newman

Newman is built on **Node.js**. Node.js is a JavaScript runtime environment. JavaScript generally runs in web browsers, but Node.js provides an environment for running JavaScript outside a web browser. With Node.js, you can run JavaScript code on servers or even directly on your own computer. This lets developers write applications directly in JavaScript, which is exactly what Newman is. One other major feature of Node.js is the package manager. The **Node.js package manager**, or **npm**, makes it easy for programmers to publish and share their Node.js code. This package manager makes installing and setting up things such as Newman dead simple, but in order to use it, you will need to install Node.js. Let's do that next.

## Installing Node.js

Since Node.js allows you to run JavaScript outside your browser, it has to be installed as an application on your computer, which you can easily do by performing the following steps:

1. Go to <https://nodejs.org/en/download/> and find the install package for your operating system.

I would recommend getting the package from the **long-term support (LTS)** section as it will have everything that you need for this book and will be a little more stable than other versions.

2. Once you have downloaded the installer, run it in the same way you would install any other program on your operating system.

You can keep all the defaults on the install wizard.

3. Once the installer has completed, verify that it has been installed correctly by opening a command prompt and typing in the following command:

```
node -v
```

This will print out the current version. You should see something like this:

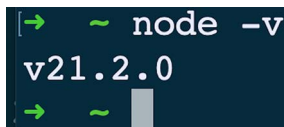
A terminal window with a dark blue background. The prompt is a green arrow followed by a tilde '~'. The command 'node -v' is entered, and the output 'v21.2.0' is displayed in green text. A second prompt is visible below the output.

Figure 9.1: Node version

Note that you might see a different version number than this. This is OK. The particular version doesn't matter too much. We are just checking to ensure that it was installed correctly.

4. Verify that npm has also been installed correctly by typing in the following command:

```
npm -v
```

This will print out the version number of npm that was installed.

Now that you have installed Node.js, you can use the package manager to install Newman.

## Using npm to install Newman

npm has a few different options for installation, but the basics of it are straightforward. If you know the name of the package that you want to install, you can simply call the `npm install` command, followed by the package name that you want to install. npm will then install that package and any dependency packages that you need in the folder you are currently in. By default, npm installs the package so that it is only available in the current folder, but there are options that let you change the scope of where it is installed. The most used of these options is `--global` (or its shortcut, `-g`), which will install the package globally. This means that it will be available to run no matter where your command prompt is currently located.

You are going to want to be able to run Newman from anywhere on your computer, so you can install it with the `-g` option by executing the following command in the command prompt:

```
npm install -g newman
```

You will see information about the different packages and dependencies that npm is installing for you flash by and in a few seconds, Newman will be installed and ready for you to use.

## Running Newman

Now that you have Newman installed, let's run a command in it to make sure everything is working correctly. The easiest way to do that is by running a collection in Postman, but first, you will need a collection to run. You can create a simple collection in Postman and then export it by following these steps:

1. In Postman, create a new collection called **Newman Test** and add a request to the collection called **Test GET**.
2. Set the request URL to `https://postman-echo.com/get?test=true`.
3. Save the request and then click on the **View more actions** menu beside the collection in the navigation tree and choose the **Export** option.

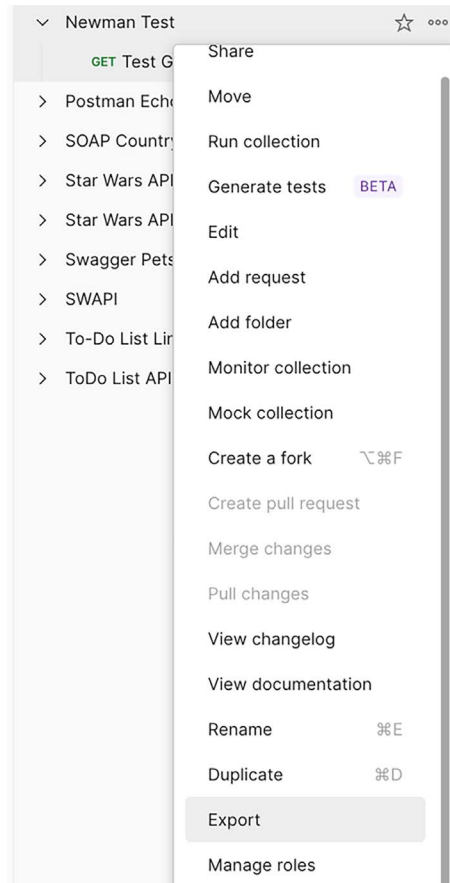


Figure 9.2: Export the collection

4. Leave the collection version as v2.1 and click the **Export** button.
5. Choose a folder on your computer to save the collection in, set the filename to `TestCollection.json`, and click on **Save**.

Now that you have a collection, you can run that collection with Newman. To do that, you will need to navigate to the folder where you have saved the collection. You can navigate to different folders while using the command prompt by using the `cd` command. This command will change directories for you, and you can use it by calling the command, along with the path to the directory that you want to go to. So, in my case, I could put in the following command:

```
cd C:\API-Testing-and-Development-with-Postman\Chapter09
```

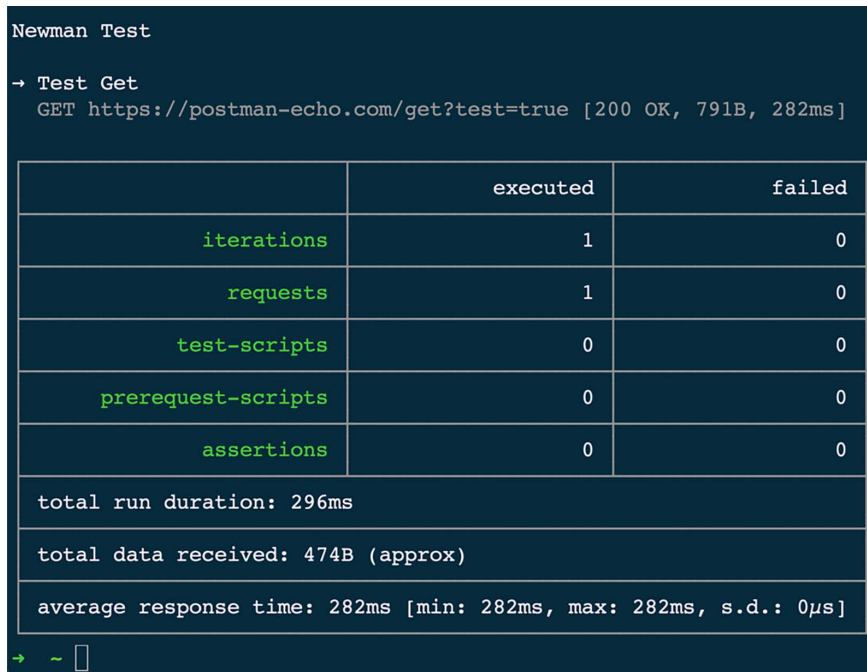
Note that on Windows if you have a space in your directory path, you will need to enclose the path in quotation marks so that your command will look like this instead:

```
cd "C:\path with spaces\API-Testing-and-Development-with-Postman\Chapter09"
```

Once you have navigated to the directory where you have saved the collection, you can run that collection in Newman. To do that, you must call the `newman` command with the `run` option and then give it the name of the file where your collection has been saved. In this case, the command would look like this:

```
newman run TestCollection.json
```

Hit *Enter* and Newman will run that collection. Once it has completed, you should see a simple report of what it ran, as shown in the following screenshot:



```
Newman Test
→ Test Get
GET https://postman-echo.com/get?test=true [200 OK, 791B, 282ms]
```

	executed	failed
iterations	1	0
requests	1	0
test-scripts	0	0
prerequisite-scripts	0	0
assertions	0	0

total run duration: 296ms

total data received: 474B (approx)

average response time: 282ms [min: 282ms, max: 282ms, s.d.: 0μs]

→ ~ □

Figure 9.3: Newman collection run report

This report shows what request was run, along with its status code and the time it was run. It also shows you details about the iterations, requests, and scripts that were run. Since this was a very simple collection, there isn't much here, but we can verify that Newman is indeed installed and set up correctly.

There are many other options in Newman for running more complex collections. You can see some inline help for those various options by calling the following command:

```
newman run -h
```

This will list the various options that you have when running collections in Newman. I won't go over all of them in this book, but some of them will be helpful for you, so I will show you how to use those ones.

## Understanding Newman run options

Now that you have Newman installed and running, it's time to look at how to use it in a variety of cases. You have already seen how to run it with a simple collection, but what about if you have environment variables or data-driven tests? In this section, I will show you how to run Postman collections, including collections that contain environment variables and data-driven tests. I will also show you some of the options that you can use to control Newman's behavior.

## Using environments in Newman

Often, a collection will have variables defined in an environment. Requests in that collection will need those variables when they are run, but the values don't get exported with the collection. To see this in practice, you can modify the Test GET request in the Newman Test collection that you made earlier by following these steps:

1. In Postman, create a new environment called `Newman Test Env`.
2. Modify the URL field of the Test GET request to set the `https://postman-echo.com` part to a variable called `base_url`.
3. Save that variable in the environment that you just created.
4. Save the request and then export the collection to a file (you can use the same filename we used previously and just overwrite it).



If you go to the command line and run that collection with Newman, you will see an error message similar to the following, since it can't resolve the `base_url` variable that you created:

```
Newman Test

→ Test Get
  GET {{base_url}}/get?test=true [errored]
    getaddrinfo ENOTFOUND {{base_url}}
```

Figure 9.4: Error resolving variable

In order to correct this error, you can manually specify the environment variable by using the `--env-var` flag. To do that, execute the following command:

```
newman run TestCollection.json --env-var "base_url=https://postman-echo.com"
```

This tells Newman that it should set the `base_url` environment variable so that it's equal to the specified URL. With that information, the collection can run correctly. However, it is a lot of work to manually type this in. If you have multiple environment variables, you can call this flag multiple times, but you would still have to type each of them in, which would get very tedious. Instead of doing that all manually, let's see how you can save yourself some work by using an environment that's been exported from Postman. You can export an environment by following these steps:

1. Click on the **Environments** tab on the left-hand side of the navigation panel, as shown in the following screenshot:

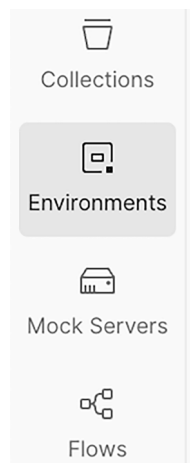


Figure 9.5: Navigating to Environments

2. Click on **Newman Test Env** in the navigation tree.
3. From the **View more actions** menu at the top-right of the environment panel, choose the **Export** option, as shown in the following screenshot:

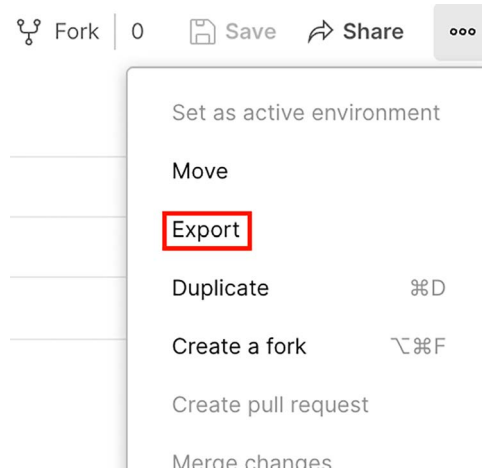


Figure 9.6: Export environment

4. Make sure to select the same folder you saved the collection in and then name the file `NewmanTestEnvironment.json` and click **Save**.

Now that you have the environment saved, you can use it in Newman. To that, you can use the `-e` flag to let Newman know where to look for environment variables. Putting this all together, you should have a command that looks like this:

```
newman run TestCollection.json -e NewmanTestEnvironment.json
```

If you run this command, you should see that it now successfully runs the collection. This is a much more scalable option than manually typing in each of the variables!

## Running data-driven tests in Newman

In *Chapter 7, Data-Driven Testing*, I showed you how to use the powerful data-driven testing concept in Postman. In that chapter, you ran the tests with the collection runner, but what if you wanted to run them in Newman? You've already seen how you can use the `-e` flag to specify an environment. Similarly, you can specify data-driven inputs using the `-d` flag. Let's modify the `Test GET` request to turn it into a simple data-driven test by changing the URL so that it has a couple of variables that you can create data for in a file. Set the query parameter and the query parameter value to both be variables.

The request URL should look something like this:

```
{{baseUrl}}/get?{{queryParam}}={{queryParamVal}}
```

Make sure you save the request in Postman and then export the collection again. You can use the same name you did previously and just overwrite the old file.

Now, you can create a file containing data that you will use as inputs. Create a simple .csv file containing two columns, each named after one of the variables. Create a couple of rows of input in the file. You should have a file that looks something like this:

queryParam	queryParamVal
test	true
query	1

Figure 9.7: Data-driven input file

Save this file as `DataDrivenInput.csv`. Before running your collection in Newman, you will need to export it again since you have made changes to it. Now that you have all the files that you need, you can run the collection in Newman with the following command:

```
newman run TestCollection.json -e NewmanTestEnvironment.json -d  
DataDrivenInput.csv
```

After a few seconds, you should see a report showing the results. You will see that it ran the request with each of the inputs in your file, along with a table summarizing the results.

## Other Newman options

The `-e` and `-d` flags in Newman will give you most of the control that you need when running collections, but there are a few other flags that I want to highlight here as well. I won't cover every possible flag in detail, but if you want, you can find out more by looking at the documentation: <https://github.com/postmanlabs/newman#command-line-options>.

If you have a large collection with a lot of requests in it, there might be times when you don't want to run the whole collection. For example, you may want to only run a subset of the requests for debugging purposes. One option you can use for this is the `--folder` option. This option allows you to specify the name of a folder in your collection that you want to run. Newman will then only run requests that are in that folder. If you want to run several folders, you can just use the `--folder` option multiple times.

Another option you may want to use is the `--delay-request` option. This option allows you to specify an amount of time (in milliseconds) that Newman should wait after a request finishes before it starts the next request. You might want to use this if you have requests that depend on each other. For example, if you had a POST request that created some data and then you were running a GET request to check whether that data was there, you might want to put in a slight delay so that the server has time to fully process the POST request.

You can also specify timeout values, which will dictate how long Newman waits for certain actions to happen before it considers them to have timed out. You can specify the `--timeout` flag to set a timeout value for how long you want to wait for the entire collection to run. You will want to be careful with this value since it can be hard to figure out how long it will make sense for this to be used if you have a lot of requests in the collection. A more targeted option for this is `--timeout-request`, which specifies the amount of time to wait for a request to complete before considering it to have timed out. You can get even more targeted than that if you want by specifying the `--timeout-script` option, which sets the maximum amount of time any of the scripts (such as Post-response pre-request scripts) can run before you consider them to have timed out. These three flags all need you to specify a number that represents the amount of time in milliseconds.

You can also use options such as `--bail`, which lets Newman know that you want it to stop running as soon as any failure occurs. By default, it will stop the run as soon as a failure is detected, but if you want it to complete the current test script before stopping, you can specify it as `--bail failure`. You can also use the folder modifier to skip running the entire collection if some of the inputs are incorrect.

Another option you may need to use occasionally is the `-k` option. This option will disable strict SSL, which is sometimes required if you are getting SSL certification errors when running your request. You can also specify the `--ignore-redirects` option to prevent the redirect links from being automatically followed.

There are several other options that Newman offers. Newman supports everything that Postman does, so if you are ever unsure of how to do something in Newman that you can do in Postman, check out the Newman documentation – you will find a way to do it.

## Reporting on tests in Newman

Newman is a powerful tool for running tests, but the point of having test automation isn't just so that you can run tests. The point of test automation is to be able to improve a product's quality based on the results you get. You need to know when tests fail so that you can follow up on those failures and see whether they represent issues in the product.

Effective test automation also requires you to pay attention to the tests themselves. You want to run tests that are effective and useful, and not just continue to run a test that isn't telling you anything.

In order to do all of this, you need test reporting. This section will help you understand how reporting works in Newman. You will learn how to use the built-in reporters that Newman has. You will also learn how to install and use community build reporters, and I will even walk you through an example that shows you how you can build your own Newman test reporter.

## Using Newman's built-in reporters

You need reports on failed tests and summaries that show you how well your tests are doing. When you ran Newman earlier, you could see some reporting that was provided at the command line. These summaries are helpful when you are running locally, and Newman has several built-in reporter options that you can use to control that report's output. You can specify which reporter you want to use with the `-r` flag. If you do not specify a reporter to use, Newman defaults to using the `cli` reporter, but there are four other built-in reporters that you can use. The other available reporters are the `json`, `junit`, `progress`, and `emojitrain` reporters.

The `json` and `junit` reporters will each save the report to a file. The `json` reporter saves the data in JSON format in a `.json` file, while the `junit` reporter saves the data in a JUnit-compatible `.xml` file. JUnit is a testing framework for the Java programming language. The JUnit XML format can be parsed by different programs, including build systems such as Jenkins, to display nicely formatted test results.

The `progress` reporter will give you a progress bar in the console while the tests run. Although I have mentioned it here for completeness, you will probably never use the `emojitrain` reporter. It prints out the progress as an emoji, but it is hard to get an emoji to display correctly in a command prompt, and there isn't any added value over what you can see with the `progress` reporter.

You can use multiple reporters at once if you want. So, for example, you could use both the `CLI` and `progress` reporter at the same time by specifying them as a comma-separated list after the `-r` flag, like this:

```
-r cli,progress
```

You can also specify additional options that will be applied to some of the reporters. For example, the `cli` reporter has a `no-success-assertions` option that will turn off the output for assertions that are successful while it is running. In order to use these options, you need to specify them with the `--reporter` (or its short form, `-r`) flag.

You will then add the name of the reporter and the reporter option to the flag, turning it into one big flag. So, for example, to use the `no-success-assertions` option for the `cli` flag, you would specify the following flag:

```
-r cli --reporter-cli-no-success-assertions
```

Alternatively, to specify that you want to save the JUnit report in a folder called `reports`, you could specify the following flag:

```
-r junit --reporter-junit-export reports
```

Note that specifying the reporter's options does not enable that reporter. You will still need to specify which reporters you want to use with the `-r` flag. There are several different reporter options available for different types of reports. You can see the list of options here: <https://github.com/postmanlabs/newman#reporters>.

## Using external reporters

In addition to the built-in reporters that Newman comes with, it allows you to use externally developed reporters. These reporters don't come installed with Newman, but once you have installed them, you can use them to create the reports that you want. Postman maintains an external reporter called *Newman HTML Reporter*. There are also several community-maintained reporters that you can install and use if you want. These include reporters such as the `csv` reporter (<https://github.com/matt-ball/newman-reporter-csv>), which will export your report in `.csv` format, and `htmlextra` (<https://github.com/DannyDainton/newman-reporter-htmlextra>), which gives you some more in-depth data output compared to the basic HTML reporter that Newman maintains.

There are also several reporters that can help you create reports for specific tools. For example, you could use the `influxdb` (<https://github.com/vs4vijay/newman-reporter-influxdb>) reporter if you wanted to be able to build Grafana dashboards from your data, or you could use the `Confluence` (<https://github.com/OmbraDiFenice/newman-reporter-confluence>) reporter to report data to a Confluence page. There are several other tool-specific reporters listed in Newman's documentation: <https://github.com/postmanlabs/newman#using-external-reporters>.

I don't have the space to show you how to use all these reporters, but I think the `htmlextra` reporter is a useful one, so I will walk you through how to install and use it. You should be able to use what you will learn next to install any of the reporters that you need.

## Generating reports with htmlextra

Similar to installing Newman, you can install htmlextra with npm. Call the following command to globally install htmlextra:

```
npm install -g newman-reporter-htmlextra
```

Once it has been downloaded and installed, you can specify it by passing in htmlextra to the `-r` option. For example, to use it with the data-driven test mentioned earlier in this chapter, you could call the following command:

```
newman run TestCollection.json -e NewmanTestEnvironment.json -d  
DataDrivenInput.csv -r htmlextra
```

Once the requests have run, you can go to the newman folder in your working directory, and you will see a `.html` file there that you can open in a web browser.



### NOTE:

If you do not see the Newman folder or if there is no `.html` file created, you may need to check on the write permissions of the folder you are in. Make sure that Newman has permissions to write files to that folder. You can usually change this at the operating system level or use elevated permissions such as those granted by using the `sudo` modifier.

When you look at the created file, you can see that it provides a nice dashboard summarizing the results of your run. The default dashboard is useful, but if you want, you can also customize it. For example, you can change the title to `My Test Newman Report` by adding the following to your command:

```
--reporter-htmlextra-title "My Test Newman Report"
```

There are several other options that allow you to customize parts of the report. You can check those out in the documentation on the GitHub repository for the Newman reporter htmlextra.

## Creating your own reporter

Even with the number of community-supported reporters out there, you might not be able to find one that does exactly what you need. In that case, you could just make your own! Doing this requires some JavaScript knowledge, but let's walk through an example so that you can see how it would work.

Unless you are fairly proficient with JavaScript, you will probably want to stick to reporters that others have made, but understanding how something works behind the scenes is still helpful. To make your own reporter, follow these steps:

1. On your hard drive, create a new directory called `MyReporter`.
2. Open a command prompt and use the `cd` command to navigate into that directory.
3. Create a blank `npm` package by calling `npm init`.
4. You will be prompted for a few things. First, you will be asked what name you want your package to have. The default name is set to `myreporter`, but this will not work. A Newman reporter must have a name that starts with `newman-reporter-`.
5. Type in `newman-reporter-myreporter` and hit *Enter* to accept the new name.
6. Hit *Enter* again to accept the default version number.
7. Type in a short description and hit *Enter*.
8. Leave the entry point as its default of `index.js` by hitting *Enter* again.
9. We won't bother with a test command, Git repository, or keywords, so you can just hit *Enter* to skip past each of those options as they come up.
10. Put in your name as the author and hit *Enter*.
11. Since this is just a test package that you are playing around with, you don't need to worry about the license too much and can just hit *Enter* to accept the default one.
12. Once you've gone through all those options, `npm` will show you the `package.json` file that it is going to create for you, which has all those options in it. You can tell it to create that file by hitting *Enter* to send the default response of **yes** to the question.

Now that you have a blank `npm` package, you can create a reporter.

13. Add a file called `index.js` to the folder you've created and open that file in a code or text editor.

This file is where you will put the JavaScript code that you will use to create your custom reporter. As I mentioned previously, creating a custom reporter will be difficult if you only have a beginner's understanding of JavaScript. I can't go into the depth of JavaScript that you will need for this process in this book, but I will show you a simple example so that you can see, in very broad terms, what it takes to do this. Newman emits events while it is running that the reporters can listen for and then perform actions based on receiving them. The following code shows a couple of examples of using those events:

```
function MyCustomNewmanReporter (newman, reporterOptions,
```



```
collectionRunOptions) {
  newman.on('start', function (err) {
    if (err) { return; }
    console.log('Collection run starting')
  });
  newman.on('item', function (err,args) {
    console.log('Ran: '+args.item.name)
  });
  newman.on('done', function () {
    console.log('all done!')
  });
};
module.exports = MyCustomNewmanReporter
```

The file needs to include a function (in this case, I have called it `MyCustomNewmanReporter`) and then it needs to export that function with `module.exports` at the bottom of the file. Once you have done that, you can do anything you want to inside the function. In this case, I am showing three different examples of how you can use Newman events.

The first event I'm looking at is the `start` event:

```
newman.on('start', function (err) {
  if (err) { return; }
  console.log('Collection run starting')
});
```

This event is sent whenever Newman starts running a new collection. You can see that I listen for that event with the `newman.on` method. I then tell it what event I am listening for ('`start`') and give it a callback that can take in two arguments. The first argument is the error object and, in this case, this is the only argument I need to specify. I then check whether there is an error and if so, stop the execution. If there isn't one, I merely log the fact that the collection run has started.

The second example in my code is listening for the `item` event:

```
newman.on('item', function (err,args) {
  console.log('Ran: '+args.item.name)
});
```

In this case, I have specified both arguments in the callback. The first one, as shown previously, is the error object, which will only happen if there is an error. The second argument is an object that contains summary information about the event that I am listening for. In this case, that object is the `item` object since this is an `item` event. This object has several properties, but in this example, I am accessing the `name` property and printing that out. The `item` event is emitted when a test has completed.

The final example in my code is listening for the `done` event, which is emitted when a collection run has completed:

```
newman.on('done', function () {  
    console.log('all done!')  
});
```

This example shows how the callback arguments are optional. In this case, I am not accessing any of the properties of the summary object and I'm not worrying about the errors, so I don't bother specifying those things.

One other thing that should be noted in this example is that the `MyCustomNewmanReporter` function takes in three arguments. The `newman` argument is the class that emits the events and these examples have shown how to use it, but I did not show you how to use the other two inputs to that function. The `reporterOptions` argument will include things such as the silent option or other reporter options that you might want to support. `collectionRunOptions` includes information about all the command-line options that were passed in.

In order to use this reporter, you will need to install it with `npm`. You can do that by first creating a package. Run this command in the `MyReporter` directory:

```
npm pack
```

This will create a zipped package that you can then install with `npm` using this command:

```
npm install newman-reporter-myreporter-1.0.0.tgz -g
```

Once it is installed, you can start to use it. Go back to the directory where you've saved the collection that we exported earlier in this chapter. You can now specify to use your custom reporter while running those tests:

```
newman run TestCollection.json -r myreporter
```

You should see something like this printed to your console:

A terminal window with a dark blue background and white text. The text reads: "Collection run starting", "Ran: Test Get", and "all done!". Below the text, there are three small icons: a red arrow pointing right, a green checkmark, and a white square.

Figure 9.8: Customer reporter output

This example is obviously very simple. If you were creating a custom reporter, you would most likely want to add more functionality to your reporter, but hopefully, this gives an idea of what it is possible to do with customer reporters. There is, of course, a lot more complexity that can go into creating your own reporter. If you want to figure out more, I would suggest that you go to the Newman GitHub repository at <https://github.com/postmanlabs/newman> and look through the documentation there. While you are there, you can also check out how the built-in reporters work. They use the same approach that you would need to use to create your own reporter. You can see the code for them here: <https://github.com/postmanlabs/newman/tree/develop/lib/reporters>. Another great example can be found in Deepak Pathania's `neman-reporter-debug` GitHub repository. If you look at the `DebugReporter.js` file (<https://github.com/deepakpathania/newman-reporter-debug/blob/master/lib/DebugReporter.js>), you will see a number of different examples showing how to use many of the Newman events and options.

Now that you know how to create your own custom reporters, we will look at how to use Newman in **continuous integration (CI)** builds in more detail.

## Integrating newman into CI/CD builds

Running your collection via the command line with Newman can help you quickly and easily run tests locally, but the real power of it comes in being able to automatically kick off and schedule runs. Many software companies now use CI and **continuous delivery (CD)** systems, which automate how builds and tests are continuously run every time a change is made to the code. Newman allows you to easily integrate into these CI/CD systems. This section will walk you through some general principles of using Newman in CI/CD systems. I will then show you a specific example of applying those principles to integrate Newman into a GitHub Actions CI build.

### General principles for using Newman in CI/CD builds

There are many different CI/CD systems and integrating with each one requires knowledge about how that system works. In broad terms, though, there are a few things you will need to do, regardless of the system you are working with:

- You will need to make sure the build system has access to the collection, environment, and any other data files that the run needs. Usually, the easiest way to do this is to include the tests in a folder in your version control system, alongside the code. Your build system will have access to the code to build it, so if you include your tests in there, it will also have access to them.
- You will need to install Newman on the CI system. If your build system has Node installed, this can easily be accomplished with npm.
- You will need to specify the command that will run the collections that you want. This works the same as running Newman in the command line locally. You just might have to be careful when it comes to figuring out where the path to the tests is on the CI server.
- You will need to set up a schedule for the tests. Do you want them to run on every pull request? Or maybe only when the code is merged into the main branch? Or perhaps you only want them to run if some other build steps have succeeded?

The exact implementation of each of these steps is going to vary, depending on the integration system you are working with, but you will need to think about each of them to run your collections in a CI/CD system.

## Example – using GitHub Actions

I will show you an example of this using GitHub Actions. GitHub Actions are a very popular tool provided by GitHub that allows you to run, build, and test commands as code is changed. They have become a very popular way to set up CI runs so let's look at how we can use a GitHub Action to automate running a Postman collection.

In order to use GitHub Actions, you are going to need a GitHub account, so be sure to sign up for one if you haven't already. Once you're ready you can get started by doing the following:

1. Log into GitHub and create a new repository.

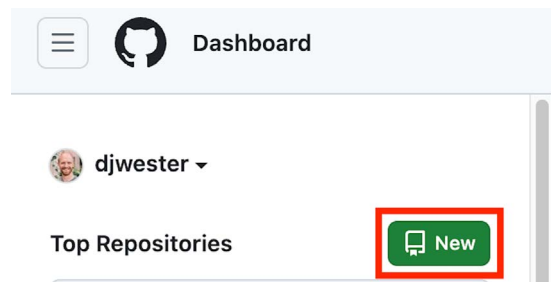


Figure 9.9: Create a new GitHub repository

2. Name the repository `postman-testing` and click on the **Create** button to create it.
3. Use the **Create a codespace** option to open a codespace that you can edit the code in.
4. Once the codespace has loaded, drag and drop the `TestCollection.json` and the `NewmanTestEnvironment.json` files into the explorer on the left panel of the code editor. This will upload those files for you.

With those files in place, you can set up a script to run these tests with GitHub Actions. GitHub Actions scripts need to be in a folder called `.github`. You can create that folder using the **New Folder** button in the codespace:

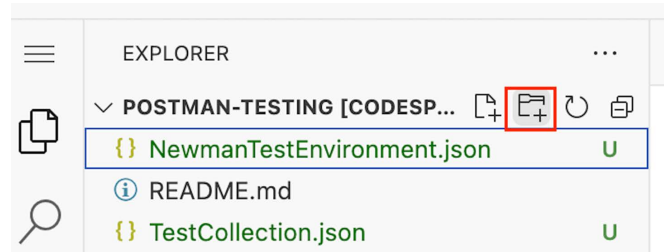


Figure 9.10: Create a new folder in the codespace

5. You will need to create another folder inside this one. Right-click on the `.github` folder and choose the **New Folder** option. Name the new folder `workflows`.
6. Click on the `workflows` folder and click the **New File** button at the top of the explorer to add a new file. Name this file `CI.yaml`.

Now, we can set up the commands in this file for installing Newman and running our tests, but before we do that, let's name this Action:

```
name: Run Newman Tests
```

Then, we will indicate what trigger we want to use to kick off a run of this file. Normally, a CI run would be triggered every time a new pull request was made or every time a new commit was pushed to the main branch, but for testing purposes, we will create a manual trigger, which we can do like this:

```
on:
  workflow_dispatch:
```

Now, we are ready to set up the commands for installing Newman and running the tests. We will do this in a job with a series of steps. We can set up the jobs section of the file and name the job run-tests. We will also specify the platform to run the tests on. In this case, I will use Ubuntu since it is a quick and free operating system to use:

```
jobs:
  run-tests:
    runs-on: ubuntu-latest
```

After that job setup, we can define the steps that we want to use (note that these steps are nested under the run-tests section):

```
steps:
  - uses: actions/checkout@v3.3.0

  - name: Install Node
    uses: actions/setup-node@v3

  - name: Install newman
    run: npm install newman
  - name: Run tests
    run: newman run TestCollection.json -e
NewmanTestEnvironment.json
```

The first step will check out the files from the repository into the CI running environment. The next step will install Node for us and then on the following step, we will use `npm` to install Newman. Once we have everything installed we can then run our tests with Newman.

With the script ready to go, we can commit the code to our repository. Normally, you would want to create a branch, commit your changes to that branch, and then make a pull request to merge that branch into the main branch. This allows others to review your code before merging it, but since we are just testing something out here, we can take a shortcut and commit these changes directly to the main branch.

In order to do that, go to the **Source Control** panel, then put in a message, and commit the changes.

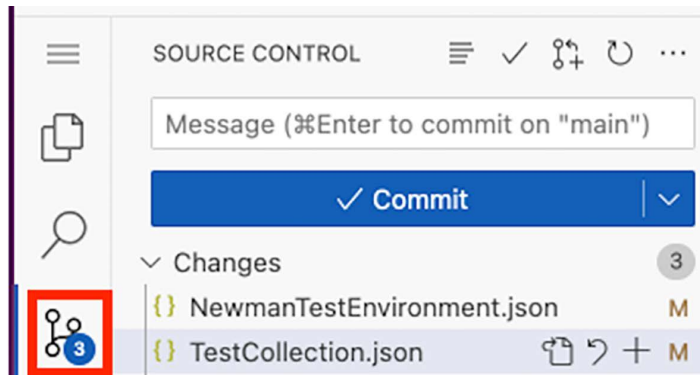


Figure 9.11: Commit changes

If you are asked whether you would like to stage all your changes and commit them directly, you can say yes and then click on the **Sync Changes** button to push your changes to the remote branch. Once you have pushed your changes, exit the codespace and go back to the repository itself and you can try running the Action we just created:

1. Go to the **Actions** tab on the top navigation:

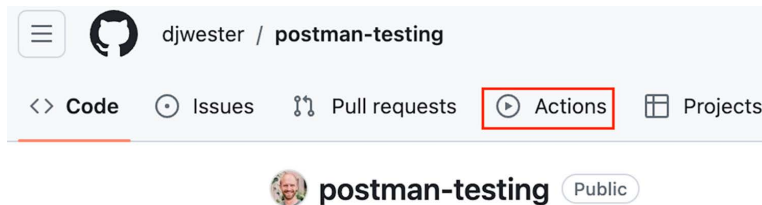


Figure 9.12: The Actions tab in GitHub

2. Click on the **Run Newman Tests** workflow in the left-hand navigation panel.
3. Click on the **Run workflow** dropdown and choose the **Run workflow** button to kick off the workflow:

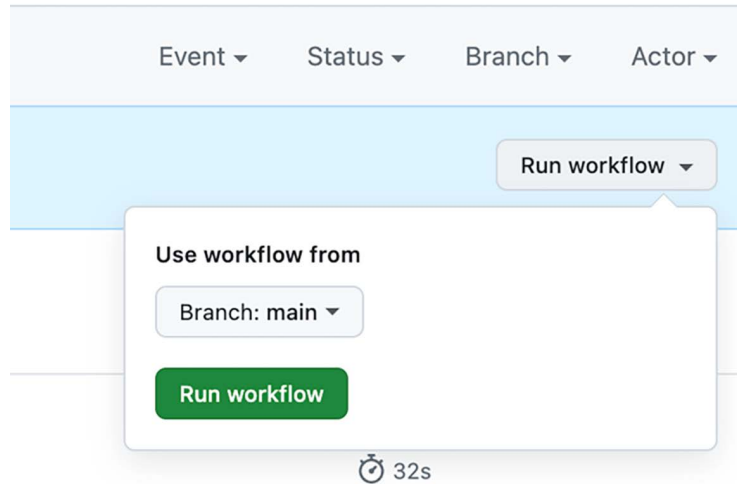


Figure 9.13: Run workflow

After a couple of seconds, this should kick off a run of your CI workflow. You can click on the link to see the results of the workflow if you want.

I have shown you a very simple example of how you can use GitHub Actions to run your tests. I hope this whets your appetite for learning more about running your tests in CI build systems. It is well beyond the scope of this book to get into the intricacies of how GitHub Actions work, but if you want to dig in further on your own, you can find the documentation for them here: <https://docs.github.com/en/actions>.

This is, of course, just one example of how to apply the general steps, but for any build system that you use, you will need to follow similar steps if you want to run your Postman collection via Newman. The exact setup and commands needed for each step will differ from platform to platform, but the general ideas will hold true.



## Summary

I don't know about you, but I just love it when you can use a tool from the command line. Newman enables you to run Postman collections from the command line, and this opens up many possibilities. In this chapter, we have learned how to use Newman to run collections on the command line and how to leverage this to integrate API testing into build systems. Of course, to do this, you needed to learn how to install Newman, as well as Node.js, so that you can install Newman with `npm`.

We then walked through some of the options for running Newman and saw how to run a collection, as well as how to include an environment and event data file for data-driven testing. We also saw some of the powerful reporting capabilities in Newman. This included the options for using the built-in reporters that come with it, as well as installing and using external reporters that others have shared. We walked through an example of using the `htmlextra` reporter. Then, we showed you how to install and use it. In addition, you learned how to create your own reporter.

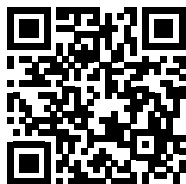
We also saw how to run tests as part of CI builds and dug into what that would look like if you were running them with GitHub Actions.

In this chapter, you have acquired the skills to use a powerful API testing tool. Put it to good use! Now that you can run tests in CI builds, it is time to look at the next step: what happens to your APIs once your clients start using them? In the next chapter, we will explore how to use Postman to monitor what is happening with our APIs as they are being used.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.com/invite/nEN6EBYPq9>



# 10

## Monitoring APIs with Postman

There are many ways to do things. Take something as simple as email inbox management. Some people read, sort, and archive every email they get. Others – like me – don't bother to sort and archive their emails but they still read every one, and still others will have inboxes with thousands of unread emails since they only look at the important ones. We all have different approaches to how we think about and handle our email, and sometimes, seeing how someone else does it can be a bit shocking to us. It's hard not to think of our way as the one that everyone uses.

This is one of the great challenges of testing. It is hard to think of ways of doing things that are different from the ways you do them. As testers, we want to prevent issues before they are released, but how do we go about predicting all the ways in which our customers will use our system? The truth is, we can't. No matter how good we get at figuring out the unknowns, we will never be able to understand and predict everything that those who interact with our software will do.

Software can also break in many ways. With an increasing reliance on cloud computing, we can be affected by a cloud provider's outage or by improper settings on a server. Applications can also break due to unexpected load or even just plain ordinary bugs. There are many ways that our systems can break.

This is why software developers and testers have increasingly turned to monitoring to help them out. Monitoring can give you information about when something is broken in your system and hence allow you to respond to issues quickly and effectively. Ideally, as testers, we would find the issues before they are released, but when customers are doing things we couldn't predict, monitoring can help us spot when something is going wrong after release.

In this chapter, I will show you how to use monitoring in Postman. We will cover the following topics:

- Setting up a monitor in Postman
- Viewing monitor results

## Setting up a monitor in Postman

**Monitoring in Postman** allows you to stay up to date with how your API is working. Normally, when you create tests in Postman, you will run them against a local build or perhaps as part of a continuous integration build. With monitors, you can run tests using staging or even production builds to make sure that certain aspects of your API are working as expected.

In this section, I will show you how to get started with creating monitors in Postman and I'll explain some of the different options that you have and when you should use them. This section will also show you how to add tests to a monitor so that you can check the exact things you are interested in.

### Creating a monitor

In Postman, a monitor is used to monitor a collection, so before we create a monitor, let's set up a collection to monitor:

1. Create a new collection and name it **Monitor SWAPI**.
2. Add a request to the collection and name it **Get People**.
3. In the **Request URL** field, enter `https://swapi.dev/api/people`.
4. Go to the **Tests** tab and, from **Snippet**, choose the **Status code: Code is 200** snippet.
5. Also choose the **Response time is less than 200ms** snippet, but then change it to be less than **400**.
6. Save the request.

Now that you have a collection to monitor, let's look at how to create monitors. To create a monitor in Postman, you will want to use the **Monitors** option on the workspace sidebar. If you do not have that option available on the sidebar, you can click on the **Configure Workspace Sidebar** option:

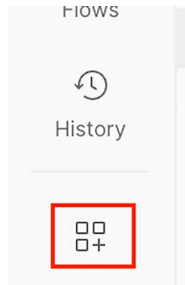


Figure 10.1: Configure Workspace Sidebar

You can then select to enable **Monitors**. Once you do that, you should see the **Monitors** icon in the sidebar:

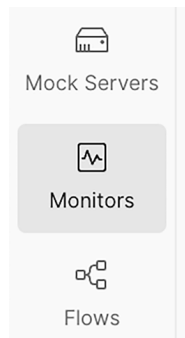


Figure 10.2: Monitors icon

Click on this icon and then choose **Create a Monitor**. You can then fill out the form to set up a monitor:

1. Give the monitor a name: **My Test Monitor**.
2. From the **Collection** dropdown, select the **Monitor SWAPI** collection.

#### NOTE:

When setting up a monitor, you have options to specify the frequency with which you want to run the monitor. Note that a free Postman account, at the time of writing this, only has 1,000 monitoring calls per month. You will want to keep that in mind when setting up the frequency of your monitor. Also, in this case, we will just be doing this for demonstration purposes, so you should delete the monitor after working through this example. We don't want to be unnecessarily running requests against a public service.



3. By default, Postman will automatically select which region to run the monitor from, but if you have specific regions that you want to run from (for example, if you want to see the latency from a specific region), you can manually select one from the list. For now, leave the **Automatically Select Region** option selected.
4. There are also additional preferences that you could set. Postman will automatically send email notifications to the email address you used to log in to your Postman account, but if you want to add additional email addresses to receive notifications, you can do so here. You can also specify options such as whether you want it to retry when it fails, or if you want to set a timeout or a delay between requests. For this example, just leave all of the options unchecked.
5. Click on the **Create Monitor** button to create the monitor. After a few seconds, the monitor will be created.

You should now be able to see your monitor in the navigation pane. If you go there and click on the monitor, you will see that you can manually kick off a run of the monitor if you want. Since there is a pretty tight tolerance on the test checking the response time, you might see a failure in the monitor if the server is a bit slow to respond. We will take a look shortly at what we can do when a monitor fails. You can also make changes to the monitor by going to the **View more actions** menu at the top-right of the page and choosing the **Edit** option:

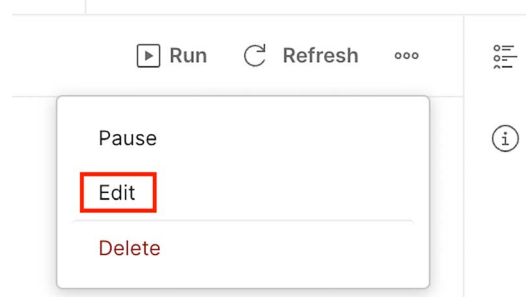


Figure 10.3: Editing a monitor

Click on the **Run** button to run the monitor; this will display a report of the run, showing whether any of the tests failed and how long it took to run the monitors. If the response code is not **200** or if the test takes too long to run, the monitor will report a failure; otherwise, it will report a success.

These steps will get you started with creating a monitor but there are several other options that we didn't use. When initially setting up the monitor, we left most of the additional settings blank. These options are available for a reason though, and so I want to take a moment to look at what they do and why they are there.

## Using additional monitor settings

There are several additional monitor settings that you can set up in Postman, as you can see in the following image:

Run this monitor ⓘ  
Check your usage limits

Week timer ▼

Every day ▼

7:00 AM ▼

Regions  
You can select one or more regions to monitor your requests from. [Learn more](#)

☒ Automatically select region  
☐ Manually select region

☒ Receive email notifications for run failures and errors  
Add another recipient email ⓘ

Stop notifications after 3 consecutive failures

☐ Retry if run fails (This might affect your billing.)  
☐ Set request timeout ⓘ  
☐ Set delay between requests

☒ Follow redirects  
☒ Enable SSL validation

Figure 10.4: Additional settings for monitors

Let's look at each of them in turn.

### Receive email notifications for run failures and errors

The email notification setting is pretty self-explanatory. Whichever email address you specify will get notified of any monitor failures. By default, Postman will have the email address associated with your account here, but you can easily add more.

You can also control how much Postman will bug you about this. If you have a monitor that is running quite frequently and starts to fail, you might get an email every 10 minutes about this failure. If you are already investigating the failure, you might find this annoying. Postman allows you to control this by specifying how many times you want to be notified about a failure that is happening every time. By default, it will stop notifying you once the same monitor has failed three times in a row, but you can change that number to be higher or lower based on your workflows.

You may want to add additional emails if you have several people or teams that are responsible for monitoring the application. For example, you might have team members in different time zones that take care of monitoring so that each team member can monitor while it is daytime for them and night for the other.

## Retry if run fails

Sometimes, there are minor glitches in our systems that we can't do anything about. If you are monitoring a part of the system where you might occasionally get this kind of glitch, you might not want to have to investigate every single one. In that case, you can use the **Retry if run fails** option. If this option is specified and a run fails, Postman will immediately run it again before notifying you of the failure. You can specify how many times it will retry (either once or twice). However, if the previous time the monitor was run, it failed, Postman will assume that this current failure is legitimate, and it will not try to rerun it for you.

Since the number of monitor calls is limited depending on which plan you are using, you will need to be aware of how this option could affect that. Automatically rerunning a monitor in this way will count against your monitor limit.

As a real-life example, I have worked with APIs that could occasionally get overloaded and refuse connections. If that happened once in a while but the server very quickly recovered, I wouldn't want a failure reported to me every time. By the time I went to look at it, it might already be running normally. In that case, I might enable the **Retry if run fails** option. However, the trick here is finding the right balance of when to retry. If this happens very frequently (say a few times a day or more), it might indicate a bigger problem that we should dig into. Retrying should be used for those cases of occasional, expected failures and not as a way to mask real problems.

## Set request timeout

The **Set request timeout** option is used to determine how long Postman will wait for a request to return data before moving on to the next request. By default, it does not specify a timeout value. Postman limits the total time that a monitor can run. On the free plan, this is 10 minutes.

If you had a monitor that was running several requests and one of them was hanging, it could prevent the other requests from running. To avoid this, you can use the **Set request timeout** option. When you choose this option, you will need to specify how long Postman should wait before it considers a request to have timed out. Of course, the amount of time that you specify here cannot exceed 10 minutes as, at that point, the total run time available for the monitor will be exceeded.

You might choose to set a timeout if you are monitoring a service that usually has a pretty fast response time. If you are making a very simple request that usually returns within a few milliseconds, a response time of even just a few seconds might mean that the server is down or having issues. You might not want to wait a full 10 minutes for the monitor to time out before you are notified of this. By setting a request timeout, you could be notified more quickly about the issues and be able to start fixing them immediately.

## Set delay between requests

Some services need a short pause between requests to allow data to be fully synchronized. If you need to do something like this in a monitor, you can use the **Set delay between requests** option to specify the time in milliseconds that you want Postman to wait after a request has finished before sending a new one.

The maximum amount of time a monitor can run is 10 minutes, so be aware that adding delays between requests contributes to this. If you make the delay too long, you won't be able to run very many requests in your monitor.

An example of when you might use this option is if you were running a simple workflow as a kind of smoke test to check that a deployment didn't have any issues. If you are running a series of requests that depend on each other, you might need to have a brief pause between each request so that all the data from the previous request can be properly processed. This kind of delay is sometimes necessary for APIs that are designed to pass UI information to the server. UI users typically can't immediately send a request; it takes a second or two for them to click on the next element in the UI. Adding a delay between requests in this kind of situation can better model how the API would be used.

## Follow redirects

Some API responses return a **redirect response**. These are responses that have a status code starting with 3. They tell the browser that it should immediately load a new URL, which is provided in the location header of the initial response.



You may want to have a monitor that checks that an API endpoint is giving back the correct redirect but might not want the monitor to actually follow that redirect. If this is the case, you can disable the **Follow redirects** option to tell Postman that you do not want it to load the redirect URL.

Some authentication workflows use redirects to send a user from the authorization server to the application server. If you are trying to monitor the authorization server, you might want to verify that you get a response, but then not follow the redirect to the main application server. This is an example of when you might want to disable this option.

## Enable SSL validation

During the development of a new API, it may not make sense to purchase a signed SSL certificate. However, if your server does not have one yet, monitors that you run against it will probably error. To work around this situation, you can deselect the **Enable SSL validation** option. This will tell Postman that you do not want to check for an SSL security certificate and so will prevent your monitors from failing due to missing certificates.

Although there certainly are times when you would need to use this option, if you are monitoring a production server, I would recommend against disabling this option. I have seen it happen that a company forgets to update its security certificate and this ends up causing errors for end users. The point of monitors is for you to know whether your end users are seeing errors and so, generally speaking, you want your monitors to fail for SSL validation errors.

Understanding these options will help you choose settings that will allow you to create monitors that do what you need them to. However, another very important aspect of a good monitor is ensuring that you have tests in place that check the things that matter.

## Adding tests to a monitor

We already added a couple of simple checks to the request that we are monitoring, but are those the right things to add? How should we think about the checks that we add to a monitor as compared to the ones we add in a regular test?

I think there are a couple of principles to keep in mind. Monitors are usually meant to be run against production code, so they should only be testing a small sample of the functionality. Don't try to add a full set of tests to requests that are being run in a monitor. As with anything in testing, this is a context-dependent decision. You need to think about why you are setting up this monitor in order to understand what kind of checks you would want to add to it.

A few common use cases for monitors might be to:

- Check that the API is up and running.
- Check that critical path workflows have no issues.
- Check for common issues.

Let's consider the first use case, where we want to verify that an API is up and running. In this case, we only need to call an endpoint and check that we get back a 200 response. We might also want to add a time check to make sure that the response comes back in a reasonable amount of time. Really slow responses might indicate that the server is under some kind of load that could lead to problems if not addressed.

There are monitoring tools that do this kind of stuff, but if your company is not yet big enough to afford them (or you don't have access to them), this kind of monitor can be a quick and dirty way to check that your API is still functioning.

On the other hand, if you are trying to check that some critical path workflow is working, the checks in your requests would probably be much more detailed. You might chain together several requests and check that the data at the end is exactly what you expect. One thing to be careful of in this kind of situation is to remember that monitors are run against production code. You want to be really careful about the kinds of things that you do in production. If you are creating new products, for example, you would probably want to delete them afterward so that you don't pollute production data.

You should also consider the fact that tests can fail. If you are running a monitor with a collection that has a few chained requests, what happens if something goes wrong partway through the run? You should make sure it is easy to go back in and clean up the data if something goes wrong.

Any monitor failures should be pretty good indicators that something has gone wrong with the system in production. They can come from code changes, although, hopefully, issues introduced in that way would be caught by tests run in your build pipeline. Often, monitors will catch things that went wrong with a deployment or perhaps with some hardware or software breaking in the production environment itself. You want to be able to trust your monitors and so you should make them fairly generic so that you can be sure that if they report a failure, it is something important to follow up on.

If you have monitors that fail for minor things, it is easy to end up with alert fatigue. In that case, you get so many alerts about failures that aren't really failures at all, that you end up ignoring the failures.

You’ve probably experienced this yourself with car alarms. They seem to go off so often that most of us don’t even pay attention anymore when one is going off. The same thing can happen with monitors. If they fail too often, you will stop trusting them. Think carefully about how you set them up.

As with all automated tests, I think it is important to *test the tests*. You don’t want your tests to fail unnecessarily, but you do need to make sure that they actually will fail if something goes wrong. For example, with the monitor we created earlier in this chapter, the timeout was set to 400ms. This might or might not have failed for you. If it didn’t, try lowering the value to something like 100 ms and running it again. You should see something like this, showing that the run has failed because it took too long:

**1 failed tests, 0 errors, across 1 region** 12:02

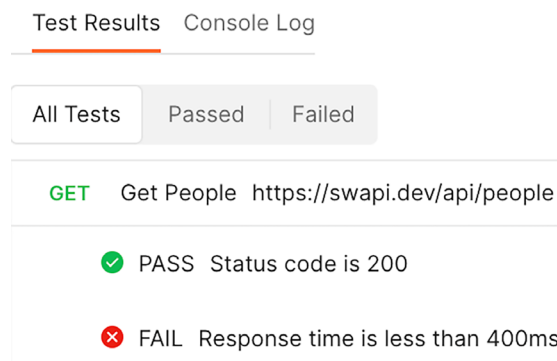


Figure 10.5: Failed check

We’ve seen how to create monitors in Postman. I’ve also shown you how to set up a monitor so that it does what you need it to. You’ve learned about the various settings that are available and seen how to add tests to monitors. With all of that under your belt, it is now time to look at how to view the results of monitor runs.

## Viewing monitor results

Once you have created your monitors, they will run on the schedule you specified. In this section, I will show you how to view and understand the results of these runs. I’ll also show you how to delete a monitor that you no longer need.

If a monitor fails, Postman will send you an email letting you know that there is a problem. This email will let you know about the failure and give you a link that will take you to the monitor. Postman will also send you a weekly summary of your monitors so that you can see how they are doing overall.

Although monitors usually run on the specified schedule, you don't have to wait for the scheduled run time. If you suspect there might be an issue, or you just want to check something out, you can manually run your monitor outside the specified schedule.

To see the monitor results, you will need to go to the **Monitors** tab and select your monitor from the list.

If your monitor has not yet run, you will see a notification to that effect, and you can click on the **Run** button to kick off a run. Once the monitor has run, you will see a chart showing some information about that monitor. The chart is split into two parts. The bottom of the chart (labeled 1 in the following screenshot) shows you how many of the tests have failed:

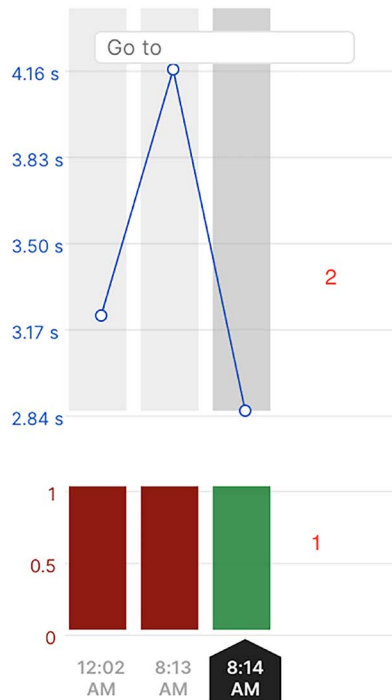



Figure 10.6: Monitor chart

In the example shown in this screenshot, one test has failed, so the label on the left shows a 1. If there were more tests in this monitor, it would show the number of tests (and how many had passed or failed)

The top part of the chart (labeled 2 in the figure) shows how long the requests took to resolve. This can be a good indicator to look at as well. You might have monitors that are not failing but if, over time, things are getting slower, that might also indicate a problem.



**NOTE:**

If you are running multiple requests in your monitor, the time shown in the monitor plot is the average of the times taken by each of the requests. If you want to see the individual request times, you can click on the **Request Split** option at the top of the chart.

If the monitor has only been run a couple of times, as shown in the previous screenshot, it is pretty easy to look through the results, but if you have a monitor that is running on a regular schedule, the number of results on this plot will start to get quite large. However, you can still view the results by using the filtering options. These options are available at the top of the chart and they can be used to narrow down the results that you see. You could, for example, look at the results from just one specific request if you wanted, or you could set the filters to look at only the results for failed or errored runs, as shown in the following screenshot:

☐ Individual requests

;

Type: All

Run result: Fai...

US (East)

☐ All

☐ Successful

☒ Failure

☒ Error

☐ Abort

Figure 10.7: Monitor result filters

One other thing that can be helpful if you are trying to figure out why a monitor is failing is to look at the test results for that monitor. If you click on one of the bars in the graph, you can scroll down on the page to see a summary of the results that show what requests were run and what checks each test did. Additionally, you can look at some results in the console log by clicking on the **Console Log** tab. This tab is also available after clicking on a run in the chart. Some details are not shown here for security reasons, but it can give you a bit of additional information.

Now that you have had the chance to set up and run a monitor as well as look at the results, I want to show you how to clean them up. Not all monitors need to stay around forever, and since the free version of Postman comes with a limited number of monitor calls, you may want to remove monitors that you don't need. Let's look at how to do that in the next section.

## Cleaning up the monitors

Since you don't need to worry about the status of the SWAPI service, I would suggest that you don't leave these monitors running. They will count against your monitor run quota and will create an unnecessary load on the SWAPI service. You can remove the monitor with the following steps:

1. Go to the **Monitors** tab and select **SWAPI Test Monitor**.
2. Click on the **View more actions** menu beside the monitor and select the **Delete** option:

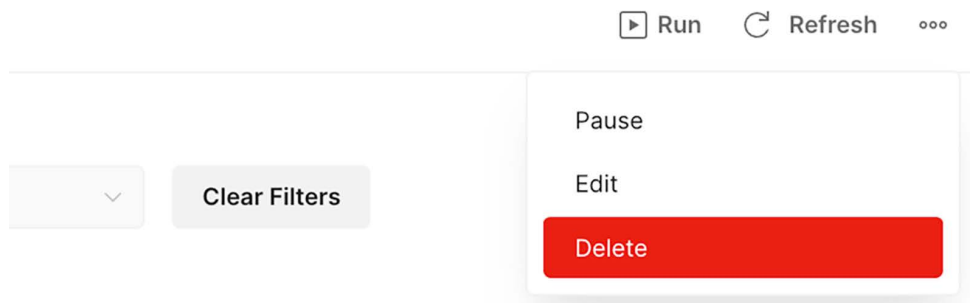


Figure 10.8: Delete monitor option

3. Click **Delete** on the confirmation dialog.

This will remove the monitor for you, and it will no longer run. In this section, you have learned how to do the final steps in the life cycle of a monitor. You can view the results and make decisions based on what you see. In addition, you have learned how to clean up old monitors that you no longer need.

## Summary

Monitoring is an important strategy in the world of modern API development. It is impossible to fully predict everything that will happen in real-world usage of our applications. No matter how well we have tested our APIs, there will always be surprises in the real world. Monitors are a way to mitigate the risk of these surprises. With monitors, we can see what is going on with our APIs and quickly respond to issues.

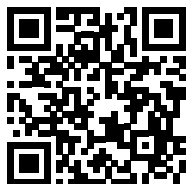
In this chapter, I have shown you how to get started with using this tool in Postman. You have seen how to create a monitor and how to add requests and tests to that monitor. You have also learned about the various options that Postman has for creating monitors and learned when to use those options. In addition, I have shown you how to view and understand the results that Postman gives when monitors are run.

Monitoring allows you to get a glimpse into how an API is working when your clients are using it, but it is no replacement for having a good understanding of how your API works. If you want to create useful monitors, you will need to know what kinds of things to look for and have a good understanding of the risks that exist in your API. In the next chapter, we will take a closer look at testing an existing API and see what lessons we can use from there to help us create better APIs.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.com/invite/nEN6EBYPq9>



# 11

## Testing an Existing API

APIs come in all shapes and sizes. I have tried to give you a few different APIs to practice on as you've worked your way through this book. If you have worked through the various examples and challenges in this book, you should have a pretty good idea of how to use Postman with various types of APIs. In this chapter, though, I want to get you thinking about how you can use Postman to test an existing API.

Although new APIs are being created every day, the reality is that there are a lot of APIs out there that do not have adequate tests. You might be working on one of those APIs. I was recently working on a project that was under a tight deadline and, to meet this deadline, we had to leave some of the API tests until later. It can be a bit overwhelming to have to go back and add tests to an API that already exists and so, in this chapter, I want to walk you through a case study where you can see examples and practice testing existing APIs. These are the topics that we are going to cover in this chapter:

- Finding bugs in an API
- Automating API tests
- An example of automated API tests
- Sharing your work

### Finding bugs in an API

Finding bugs can be easy. Sometimes they just jump right out at you. However, the easy-to-find bugs aren't the only ones that matter. Good testers are those who can find all the bugs that matter.



Doing this is often difficult and takes some practice. In this chapter, I will help you to get started with creating good automated tests, but before going deep into that, you should get some practice in the skill of looking for bugs. In this section, I will help you to set up an API locally that you can use for testing.

I will then show you how to do some testing and exploration of this API and help you find some bugs in it. I will also show you the steps to find one specific bug in the system so that you can get a bit of an idea of how to go about doing this for yourself. Let's dive into this by setting up the API that you are going to test.

## Setting up an API for testing

There are a lot of sites that you can practice calling an API on. However, many of them do not give you the ability to practice POST, DELETE, and PUT calls. This is understandable since those calls can change things for other users in the system. However, it would still be beneficial to be able to test on a fully-fledged application. Most APIs do not exist in isolation and there is often a user interface where you can see the effects of the API calls you make. Having an application with a user interface will help with understanding what the API is doing and when it might be doing something wrong. To help you out with that, I have created an application that you can use to practice your API testing with. To get started with using this application, you will need to make sure you have a few things in place.

We will be using the same application we set up in *Chapter 1*. You can refer back to the installation directions in that chapter if you haven't yet set it up, or you can follow the instructions in the readme of the repository for this application (<https://github.com/djwester/todo-list-testing?tab=readme-ov-file#install-and-setup>). I would recommend using GitPod to run it, but you can set it up to run locally if you want.

Once you have the application running, you can start making requests to the server. This application provides a basic API for creating and editing todo list items. It comes with a basic user interface that you can see by putting the URL in your web browser's address bar. You can also see documentation on the various endpoints by going to `{your-url}/docs`.

This API is a full CRUD API, so you can use GET, POST, PUT, and DELETE with it. We will use this API throughout this chapter as we learn more about testing an existing API.

## Testing the API

The first thing you should use this API for is to practice finding bugs. Later in this chapter, we will look at using it to practice setting up automation in Postman. To get started with exploring, though, you will need to first have some familiarity with how this API works:

1. Ensure that you have the server running and then create a new collection in Postman called something like `ToDo List API`.
2. Add a variable to the collection called `base_url` and set its initial value to the value of the URL for your application. If you set it up locally, this will be `http://127.0.0.1:8000`, but if you set it up with GitPod, it will be the semi-random URL generated by GitPod (which you can find on the **Ports** tab).

One thing to be careful of if you are copying the URL from GitPod or the browser is that it might copy it with a trailing slash on the end of the URL. When creating the `base_url` variable make sure you remove that trailing slash from the end, or you might end up with some unexpected results.

3. Make sure to **Save** the collection.
4. Add a request to this collection called `Get Tasks` and set the **URL** of the request to `{{base_url}}/tasks`. Make sure the request method is set to **GET**.
5. Send the request and you should get back a list. If you have not yet created any tasks, it will be an empty list.
6. Create a new task by changing the request method to **POST**, and on the **Body** tab, by setting the type selection to **raw** and then choosing the **JSON** option from the dropdown.

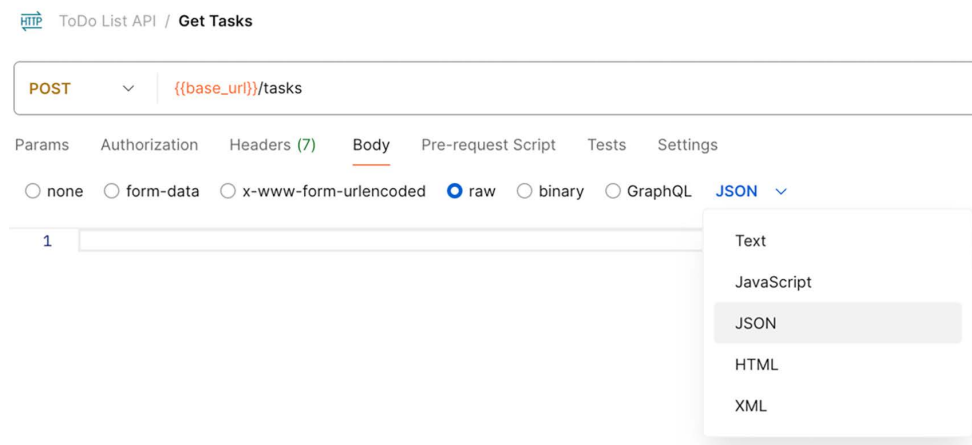


Figure 11.1: Set the body type to JSON

You can then fill out the actual body of the request using the code below. You don't need to specify the ID since the server will automatically create it with the next available ID, but you can set the description, status, and created\_by fields by specifying JSON similar to that shown in the following code snippet:

```
{
  "description": "Read this book",
  "status": "Draft",
  "created_by": "user1"
}
```

7. Send the request and you should see that a new task has been created.

You can also check that it has been created by changing the action to GET and sending the request again to see the full list of tasks in the system.

Similarly to creating a new task, you can use PUT to modify an existing task. To do that, you can use the following steps:

1. Look at the full list of tasks (by calling GET on {{base\_url}}/tasks) and get the ID of the task you just created.

When you are modifying a resource, you need to let the server know exactly which resource you are modifying, which you do by specifying the task ID in the endpoint. The task ID is the resource identifier that tells the server which task we are interested in.

2. Set the API action to PUT and add the task ID to the end of the URL.

It should now look like {{base\_url}}/tasks/1 where the 1 is replaced with whatever task ID your task has. As with creating a new resource, you need to specify the data that you want to modify. You do that in the body of the request.

3. Set the **Body** type to **raw** and choose **JSON** from the drop-down menu.
4. Specify which fields you want to modify. For example, to modify the body of the task, you could specify JSON as shown in the following snippet:

```
{
  "description": "Read this book carefully",
  "Status": "In Progress",
  "created_by": "user1"
}
```

Notice that, although you are only modifying the data in the body field, you need to specify all of the other fields as well. This service is very simple, and so it will blindly apply whatever you give it in a PUT call. It will not just update the fields that you request but rather will replace the entire resource with the data you have given it. This service does not support it, but if you are interested in only changing one property (as we are doing here), you can sometimes use the PATCH method instead of PUT. PUT generally updates a resource by replacing it while PATCH is intended to only modify the specified fields and leave everything else the same. For this book, we will continue to use PUT, but if you see PATCH used in production, just think of it as something similar to PUT but that does a partial update of the resource instead of completely updating it.

In order to delete data with this API, you need to call the endpoint for the resource that you want to remove and specify the action as DELETE. However, the API is set up to only allow the user that created the task to delete it. In order to tell the API what user we are, we need to use a token. You can get a token with the following steps:

1. Create a new request called `Get Token` with the **URL** set to `{{base_url}}/token`.
2. Change the request method to **POST**.
3. On the **Body** tab, choose the **x-www-form-urlencoded** option.
4. Create a new key called **username** and set its value to `user1`.
5. Create a second key called **password** and set its value to `12345`.
6. Send this request, and in the response body you should see an **access\_token** field. Copy that token.

You can now take the token to the request where you are trying to delete the task and on the **Authorization** tab, select the **Bearer Token** option from the **Type** dropdown. Paste the token that you copied (without the quotation marks around it) into the **Token** field. You should now be able to send the request to delete the task that you made previously.

Now that you know how to call this API, it's time to look at what bugs you can find in it.

## Finding bugs in the API

This API is not very robust. It doesn't do much error checking and so it should be pretty easy to break it. Before moving on with this chapter, I want you to do just that. Spend some time exploring how this API works and trying to break it. There are several possible ways to do that, and I would strongly encourage you to spend a few minutes thinking about the kinds of inputs and configurations you could use that might cause problems for this API. See if you can get the server to give you a 500 error!

Note that this is meant to be a challenge, which means that it is OK if you find it a bit tough to do. Stick to it for a while and you might just be surprised at what you can do. It's OK to feel a bit frustrated and lost sometimes as that is often when the most learning is happening! I will share with you one of the ways that you might encounter an error later on in this chapter, but first, see what you can find on your own.

As a reminder, when you are looking for bugs, don't just try the things that you expect to work. Think about what you could try that might show you places where the API doesn't work, such as things like the following:

- What kinds of inputs to the system might break things?
- What would happen if I tried to modify or change objects that don't exist?
- Are there any invalid or unexpected ways I could interact with this API?
- What are some unexpected things users of this application might do?

Add your own thoughts to a list like this. What kind of things can you think of that might make this API break? This API is meant to be broken in various ways, so see what you can do to find those broken spots. Next, since this API is meant to break, let's look at how you can reset it if it does break.

## Resetting the service

As you try things out in this API, you might end up causing corruption that the API can't recover from. If you do that, don't worry—it's part of what you are trying to do. However, it would mean that you can't do any further exploration and testing, so you may want to reset the service. You can do that with the following steps:

1. If you are running the application locally, go to the Command Prompt that the service is running from. If you are running it in GitPod, go to the **Terminal** tab of the VScode instance running in your pod.
2. Hit `Ctrl + C` to stop the service.
3. Restart the service by calling `make run-dev`.

If things are still broken, you might have corrupted the database itself (good testing if you managed to do that!). In that case, you can reset the database itself.

4. Once again hit `Ctrl + C` to stop the service.
5. Call `poetry run python remove_tables.py` to reset the data.
6. Once again restart the service by calling `make run-dev`.

Now that you have had the chance to do some exploring and find some bugs on your own, it's time to look at an example of the kinds of bugs you can find in this API.

## Example bug

There are several bugs that you might have found in this API. I will share the steps to reproduce one of them here so that you can get a better feeling for how to find a bug like this. Finding bugs is an exploratory process, so rather than going straight to how to make the bug happen, I'll take a bit of a roundabout route to demonstrate how you might discover bugs like this.

Let's start with trying different invalid inputs on a task. You might try putting in invalid IDs, like maybe doing a GET on the ID of a task that doesn't exist, or trying to get a task using a negative number or some other invalid input. You might then wonder whether the database has cleared everything up correctly, so you try deleting a task and then doing a GET on that same ID. Everything probably seems fine up until now, but you decide to try to "modify" the task that has already been deleted. Using the same ID that you just deleted, you change to a PUT request and fill out the body.

At this point, if you send the request, you should get back a 500 error from the server. Now, what's interesting about this is that you have built up a narrative in your head: the reason you got this error is because you were trying to modify an object that had been deleted. But when you stop and think about it, you realize that the problem might be even bigger than that. Perhaps it just breaks any time you try to do a PUT using an object ID that is not in the database. You try with another ID that you know has not been created, and sure enough, you get the same error. Well, congratulations! It looks like you've managed to find a bug.

Exploring an API and trying to find bugs interactively is an important testing skill, but often when you are working with an existing API, you will be interested in adding test automation. It is often important to be able to run a suite of tests against an API every time changes are made, so let's turn our attention to the process of creating automated tests.

## Automating API tests

API test automation is often valuable, and in this section, I want to help you think about how to get started with creating test automation for an existing API. However, I'm not going to walk you through a step-by-step process of doing this since I want you to practice doing this on your own. You will learn more if you try some things out on your own. I won't just throw you in the deep end though. In this section, I will give you a quick review of some concepts that will help you with creating API test automation. I will then give you a challenge that I want you to try, and I'll also give you some guidance on how you might break down your approach to solving this challenge.

If you work through this challenge, you will find that you have learned a lot about creating good API test automation. So, let's start with a review of some important concepts for API test automation and think about how they could play out in creating tests for this API.

## Reviewing API automation ideas

We have already looked at exploring this API and so you should have a pretty good feeling for how it works and what the various pieces of it are. Use that information to help you to put together a well-organized collection of tests. Think about how you are going to organize the tests and how you can set things up to test the various endpoints and available actions for each one.

You will want to think about how to create test validation. What kind of things do you want to check for each API call? What kind of inputs might you want to use to check these things? Obviously, this application is very simple, but try to think about how someone might use a todo list app. What kinds of things might they try and what things would you want to check for something like this?

In this section, I want to challenge you to take this existing API and create a suite of tests for it. That is a pretty big task, so let's take it on one piece at a time. I want to strongly encourage you to do each of these pieces for yourself. Save all of your work into a collection and at the end of this chapter, I will show you how you can share it so that others can see what you have been learning. I will work through examples showing how I approached these challenges, but you will learn much more from this if you first try to work through the steps on your own.

This is a big challenge, but everything in the book up until now should have prepared you for this, so I'm sure you can do it! I will also include a few hints and ideas for you in the next sections that will help you to break down your work and approach it systematically.

## Setting up a collection in Postman

I would suggest that you don't just start adding tests right away. First, take a bit of time to think about how you want to set up the collection in Postman that you can use for your testing. Careful consideration of how you are going to organize your collection and set up the tests will help you out a lot later on when you are adding tests. It is also a good practice to think this stuff through as it can have a big impact on how easy it is to run and maintain tests later.

Here are some things that you might want to consider when defining the structure of your collection:

- What are the various endpoints for this API?
- How does the testing of the different parameters fit into the way you would set up the tests for this collection?

- How are you going to test the various types of actions that are available?
- In what ways do the different pieces of this API relate to each other and how might you want to check those relationships in your tests?

You may want to think through these ideas and map out some of them on a piece of paper first so that you get a good idea of how you are going to approach this. This might feel like it is overkill for this API, and it probably is, but it is a good skill to practice. Real-life APIs will often be much more complex than this and being able to break them down and think carefully about how you will structure the automation is a crucial skill to have. This structure will be helpful when you start adding tests to the requests.

You can find out a lot of this information by exploring the API (either in Postman or with the UI). You may also find it helpful to look at the documentation, which you can find by going to `{{base_url}}/docs`.

## Creating the tests

Once you have the structure in place, you can start to flesh it out and create the tests themselves. You will want to fill in the appropriate URLs for each request, being sure to create parameters wherever it makes sense to do so. You will also want to consider whether there are any places where you could use data-driven testing to simplify your tests. As you create the tests, here are a few questions you might want to consider:

- What kind of things do you want to check for each request?
- How can you reduce the number of inputs to the test while still getting good coverage?
- Are you using variables and environments efficiently?
- Will these checks be stable and consistent?
- Will these checks actually fail if something goes wrong?
- What will happen when this test is run multiple times?

Creating tests for each of the requests is a big job, but it is a critical part of any test automation work. The things that we check for inside each test will, to a large extent, determine how useful the tests are. Take a bit of time to work through this challenge. You will find that as you try to balance out the need for coverage with the need for some degree of efficiency, you will learn a lot. Make sure you also take the time to think about how you can share data between tests. Environments and variables are powerful tools in real-life API testing so be sure to use this challenge as an opportunity to test your skills on these things.



Don't rush through this challenge. There is a lot that you can do with this, and by spending time working through it, you will be learning a lot. I keep repeating this throughout this book, but that's because it is true; the best way to learn this kind of thing is by practicing. Take your time and think about how you are approaching test creation. Try different things until you are comfortable with the structure of your request and with the tests that you've added to them. Once you have put in the effort to do this yourself, it will be helpful to compare what you came up with to another solution. I worked through this challenge myself and I have shared my approach in the next section.

## **An example of automated API tests**

I've laid out a challenge for you around creating test automation for a simple API. I hope you have worked through that challenge on your own and have a robust suite of tests that you can run against this API. In this section, I will show you one possible way that you could go about doing this. This is by no means the only way to do this and perhaps is not even the best possible way. If what you have looks quite different than this, that is no problem at all. There is a lot of benefit to seeing how other people approach their work, so treat this section as a place where you can see the work of someone else and perhaps even learn something from it.

In this section, I will walk you through the process of designing the collection layout, and then I will show you the various tests that I made and how I set things up to share data between them and make them easy to run and maintain. I will also explain my thought process and why I decided to do things the way I did. This will allow you to see my considerations and help you to think about the kinds of trade-offs that go into designing API test automation.

## **Setting up a collection in Postman**

The first task was to figure out how to set up the collection in Postman. I approached this by first creating a collection called `ExampleToDoListTests` to store my tests in. Before I started creating any requests in there, though, I decided to diagram out how the API worked and how the various pieces of it related to one another. To do this, I spent some time looking through the documentation and exploring how the various endpoints worked. I didn't just want to figure out what endpoints were available. I wanted to also determine how the different data in the system related to each other and so I drew out a diagram showing this. In real life, it is possible that the system you are testing already has diagrams showing the relationships of databases or other objects. If this is the case for you, it would be a good idea to use those to inform your understanding of the system, but I think it is still helpful to try and draw something yourself.

The act of putting it down on paper (or a computer screen) yourself helps you get a better grasp of how the system works. In my case, the diagram I drew looked like this:

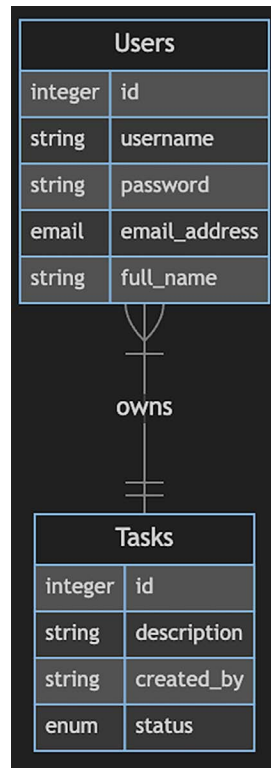


Figure 11.2: API layout

Don't worry too much about what all the symbols on this diagram mean. This is known as an **Entity Relationship Diagram (ERD)**. This type of diagram is used to show how different entities relate to each other. It is often used to represent databases. If you are not familiar with this kind of diagram just know that it shows what data is stored in each endpoint and how they relate to each other. It tells us that there are two main data types in this application; users and tasks. A user can create and own multiple tasks, but each task can only have one user that owns it. The diagram you made certainly doesn't need to be an ERD, but hopefully you drew something out that helped you understand how the application is set up.

Back to my case, now that I have a good grasp of the API structure and relationships, I can look at the actions available on each endpoint. I pulled this information from the documentation and also from some exploring I did where I discovered some undocumented endpoints.

I put together a small table to help me map all of this out, as you can see in the next figure:

Endpoint	Actions
/token	POST
/tasks	GET, POST
/tasks/<id>	GET, PUT, DELETE
/user	GET, POST
/user/<id>	GET, PUT, DELETE
/user/me	GET
/user/admin	GET
/tasks/<id>/draft	PUT
/tasks/<id>/in-progress	PUT
/tasks/<id>/complete	PUT

Figure 11.3: API actions

The last three endpoints in this list are not shown in the documentation. I found them by looking at the developer tools in my browser while interacting with the user interfaces. When I am doing API testing, I like to explore the app with the **Network** tab of my web browser open. This allows me to see what kinds of API calls are being made. Sometimes, developers forget to add documentation for an endpoint, and I can find new ones that way. I can then follow up on getting them documented for future users.

Understanding the structure of the API is a good place to start, but it is certainly not the only thing I want to consider when designing my test layout. An API serves a purpose. When thinking about the purpose of an API, I want to consider what workflows might be important for the users of the application. For example, it might be important to test a workflow where someone creates a task and then realizes that they were not logged in and so logs in and then edits the task. In bullet-point form, this might look like calling the following set of endpoints in this order:

- POST /token: As anonymous user
- POST /tasks: To create a task as the anonymous user
- POST /token: As user1
- PUT /tasks/<task-id>: To modify the task that was created in the first step

If I create tests for this kind of workflow and also add tests for each endpoint, I might end up duplicating some things that I'm checking. After some consideration, I decided that I would still create a folder for each endpoint. I would use those checks as **smoke tests** to let me know that at least the basic functionality of the API was still working.

Smoke tests are a set of tests that can run quickly and give you an early indicator of any potential problems. Much like the saying “where there is smoke, there is fire,” smoke tests let you make sure that everything is reasonably stable before you go on with more complex testing. You don’t want to have too many smoke tests, but having a set that calls each endpoint once is good practice.

Once I pulled all of these pieces together, I ended up with a folder structure that looked something like this:

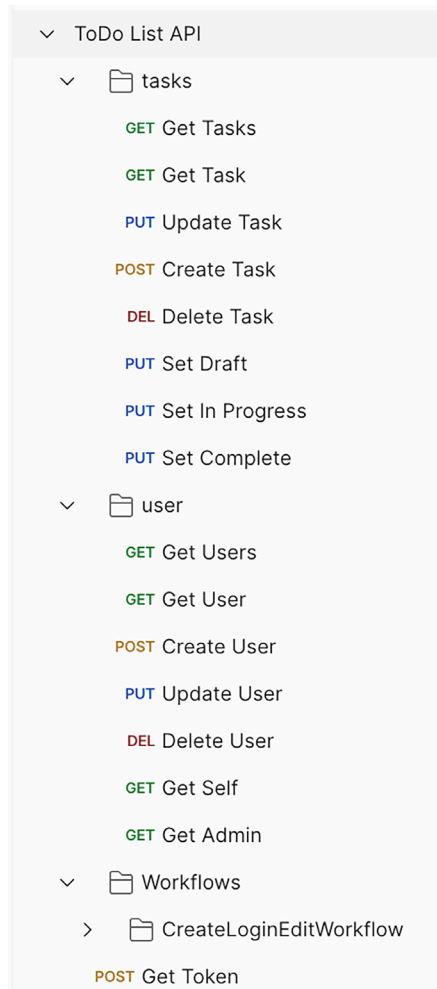


Figure 11.4: Test folder structure

The request to get the full list of tasks has some query parameters on it. You will notice that I have not included separate requests for each of the query parameters. Instead, I plan to use data-driven testing to exercise those different options. I will get into that in more detail later in this section.

I included an example of the kind of workflows you might check with something like this. In real-life applications, workflows would be built out based on conversations with project managers and others in the business along with the developers and testers. I included one here as an example, but I'll be focusing on the other parts of this collection in more detail and include those just as examples of the kinds of things you might think of when doing this kind of testing. A collection with a bunch of requests is a good start, but it isn't worth much without some tests.

Creating the tests

Now that I have the structure of my requests in place, I'm ready to start creating them. The first thing to do is to fill in the endpoints for each of the requests that I've made. I'm going to need to go through each one and add the URL, but I want to be forward-thinking as well. I'm working on a local build here. What if, in the future, I needed to test this on a production server somewhere or even just on a different port number? I don't want to have to go through and update every single URL in this test suite. This is where environments are helpful.

Updating the environment

To make future maintenance reasonable, I will make a variable to store the base URL for this collection. I anticipate that I'm going to need other variables as well, so I will go ahead and create an environment that I can use to store the variables, as shown in the following screenshot:

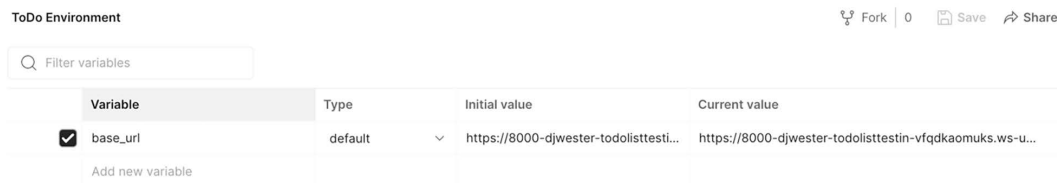



Figure 11.5: Creating an environment

As you can see in the screenshot, I have also created a variable called `base_url` that I will use in the URL for each of the requests.



**NOTE:**

As shown in the previous screenshot, environment variables have an initial value and a current value. With a newly created environment variable, those values will both be the same, but if you modify the value later, Postman will set that in the current value while keeping the initial value there for your reference.

Once I have added the environment and made sure that it is the active environment, I can enter the request URL for each of the requests. For example, the URL for the GetTasks request would look like this:

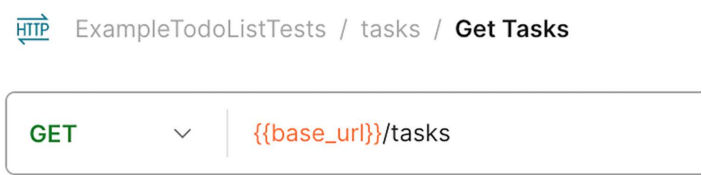


Figure 11.6: Request URL

Using a similar pattern, I can quickly add the URL to each of the requests. I also add a variable to the environment for the **ID** of the task, as shown in the following image:

Q Filter variables

	Variable	Type	Initial value	Current value
<input checked="" type="checkbox"/>	base_url	default	https://8000-djwester-todolisttesti...	https://8000-dj
<input checked="" type="checkbox"/>	task_id	default	1	1
	Add new variable			

Figure 11.7: Task ID in the environment

This will allow me to use the same request to potentially call more than one task. While going through and adding the URLs, I also updated the call methods so that they are using the correct one for each request. Now that this is all filled out, I'm ready to start adding tests to these requests.

## Adding tests to the first request

I know that I will most likely want to share some of the test automations and scripts that I make across multiple tests. However, a common mistake that automation engineers (and developers in general) make is to overgeneralize – they often spend too much time making a general solution that will work in a wide variety of contexts. This is great, except that it can take a lot of time to make a generalizable solution like this, which can lead to wasted time and effort. To avoid that, I will first add a test to one of my requests and then as I start adding more tests after that, hopefully, I will be able to see which things it makes sense to generalize and share between tests.

The first place I will add tests to is the `GetTasks` request. I could just check that the entire response body does not change. This is an acceptable thing to check if you are pretty sure that the data will be stable. However, if you expect that it might change, you will want to be more careful when checking something specific. In this case, since this is a list of requests, I fully expect that it will change over time, so I don't want to just blindly check it.

I could instead set up a test that ensures that the first task in the list doesn't change. If this was the type of data that I would expect to stay static that would be helpful, but in this case, is it really reasonable to expect that nobody will touch it? Well perhaps if the task was completed, but otherwise we would have to assume that it will change. Ultimately, there are two things I want to check here. I want to check that this gives back a list of tasks, and I want to check that the tasks have the correct data. The fact that it is a list can be checked implicitly. If I assume that the result is a list and use an index to extract items from the list, the test will fail if the result is not returned as a list. I don't need to explicitly add a test for that. I don't know how long this task list might get in the future, so I don't really want to loop over every item in the list as that might slow things down, so I will add a test that just checks the first item in the list. I will start with a naïve implementation here that assumes that the first task in the list doesn't change and then later we'll think about how we could make this test more robust.

The test code shown below reflects the state of the app on my computer. You will notice the assumptions that I am making here that mean that this test won't work on your computer. You probably won't have a task with the same ID and description as mine as the first task in your list. However, we will use it as an example for now and look at how to make it more generally applicable later:

```
var jsonData = pm.response.json();
var firstTask = {
  "id": 77,
  "description": "Learn API Testing",
```

```
    "status": "Complete",  
    "created_by": "user1"  
  }
```

I can then create a test that will check that the data for the first task stays consistent:

```
pm.test("Check first task data", function () {  
  //assume that the first task won't change  
  pm.expect(jsonData[0]).to.eql(firstTask);  
});
```

Now, this is a good start, but as I said, this test isn't robust to any changes. If anything in the task is updated, this test will fail, and we probably don't want it to. There is no business reason to care about the exact description or ID of this task. I don't like having to go and update tests multiple times just because of some small change elsewhere in the system, so let's see if we can make this check a bit more robust. To do that, I will create a test that checks that the tasks have the correct fields in them. I can do that with a test like this:

```
pm.test("Check that the first task has required fields", function () {  
  var taskKeys = Object.keys(jsonData[0]);  
  pm.expect(taskKeys).to.have.members(['id', 'description',  
    'status', 'created_by']);  
});
```

This test gets all the keys from the first task in the list and checks that those keys are the ones we expect. This is a much more robust check. If you have created at least one task and you run this test against your version of the app, it should pass for you. So, now that I have a test in place for this first request, let's look at some of the other requests and see whether there is anything we can do to share data between the tests.

## Adding tests to the second request

The second request I will add a test to is the GetTask request. This request will get the information about a single task rather than the full list of tasks, but don't forget that we've parameterized the task ID for this. This means that we can't assume that we are going to get one specific task. I can't make a test that checks the task data, but I should be able to do a similar test to the one where I check that the task has the required fields. In fact, it seems like I should be able to directly use that same test in both requests.



Instead of having the same test exist in two different requests, I will move it out of the GetTasks request and into the tasks folder so that both requests can use it. This can be done with the following steps:

1. Highlight the text for that test in the **Post-Response** panel on the **Scripts** tab of the GetTasks request and cut it, as shown in the following image:

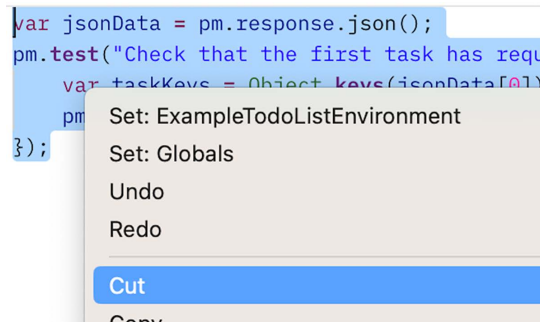


Figure 11.8: Cut the test out of the first request

2. Click on the **Edit** option on the menu beside the tasks folder, as shown in the following screenshot:

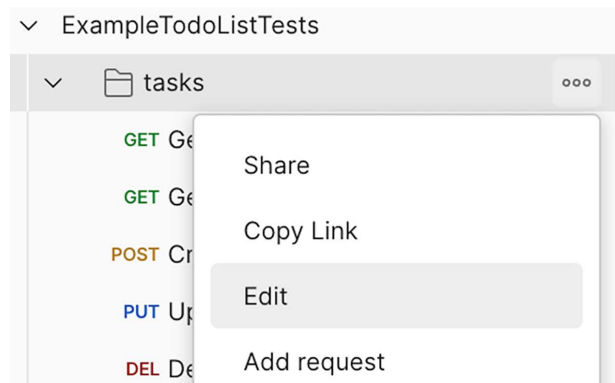


Figure 11.9: Editing a folder

3. Go to the **Post-request** panel of the **Scripts** tab for the folder and paste in the copied test.
4. Click on the **Save** icon at the top-right of the panel to save this test into the folder.

The test is now saved in the folder instead of the request. This test will now run against all of the requests in the tasks folder. For example, if you go to the `GetTask` request and run it, you will see that the test is run too. In fact, if you go and do this, you will see that the test fails with an assertion that looks like this:



Figure 11.10: Failing test

This is an odd error. Maybe we should dig into it. If you go over to the body tab of the response, you can see that it only has two fields, the **description** and the **status**.

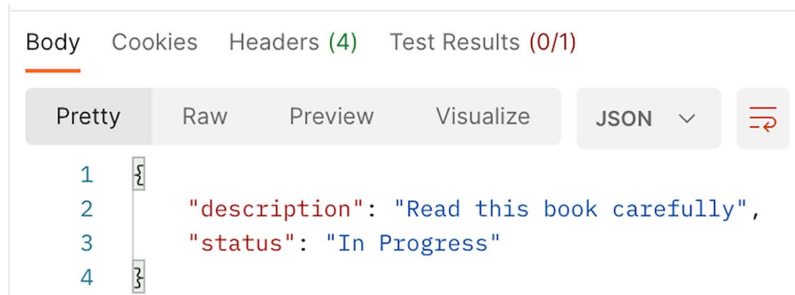


Figure 11.11: Single todo result

It doesn't have the `id` or the `created_by` fields. This seems a bit strange. Why would we show all that information in the list of results, but not with the individual results? I would say that in this case, our tests have found a bug! I guess we should leave that test as failing for now, since it found a bug. Let's move on to adding a test for creating new tasks.

## Adding tests to the POST request

Before adding tests though, we should fill out the body of the request that will be sent during the POST call. On the **Body** tab of the request, I will set the type to **raw** and then select **JSON** from the drop-down menu and add the following body:

```
{
  "description": "Do Something",
  "status": "Draft",
  "created_by": "user1"
}
```

Now let's run this request and see what happens. It creates the task, but on the **Test Results** tab it shows the same error we had before. When we saw the error on the **Get Task** request we assumed it was because the structure of that response was different, but the response for this request looks correct. It has all the fields we would expect. So, what is going on here? Well, let's go back to the test definition on the folder. This test is checking that the first task has required fields. So, it is getting the first task in the list and then checking the field on that task. When we create a new task, we don't have a list of tasks, but rather just one task. So, is it still possible to share this test across the folder? Well, yes! We can use Postman's package library. This is a fairly new feature in Postman and so in order to use it you will need to make sure that you have updated to the latest Postman version.

What we need to do is to turn this into a shared function. On the right-hand panel, you can click on the link to **Open package library**. Name your package `common-tests` and create a function in there that looks like this:

```
function checkTaskFields(task) {
  var taskKeys = Object.keys(task);
  pm.expect(taskKeys).to.have.members(['id', 'description', 'status', 'created_by']);
}
```

Now add a call to make this function available:

```
module.exports = { checkTaskFields }
```

Delete the other code from the **Post-response** panel on the folder. Now we can use this function in different requests:

1. Go to the **Post-response** section of the **Scripts** tab of the `Get Tasks` request.

2. Go to the **Find & use packages** dropdown on the right.
3. Select the `common-tests` package

Postman should automatically add and import what looks something like this to the top of your script:

```
const commonTests = pm.require('@interstellar-sunset-3010/common-tests');
```

Note that the first part of the require will be different for you as it will use your Postman identifier. You can now add the following code to call this shared function in a test:

```
var jsonData = pm.response.json();
var task = jsonData[0]
pm.test("Check first task field", function () {
    commonTests.checkTaskFields(task);
});
```

This will get the first task in the list and pass it to the shared `checkTaskFields` function. Similarly we can use the same shared function in other tests. On the create task request we can import the shared package and then just directly pass the response to the shared function like this:

```
var jsonData = pm.response.json();
pm.test("Check task fields", function () {
    commonTests.checkTaskFields(jsonData);
});
```

## Cleaning up tests

Now that we have seen how to share functions, let's look at test clean-up. You usually want your tests to clean up after themselves. There are a couple of reasons for this. First of all, when you re-run tests, you want to be able to start from a clean slate, and secondly, if you don't clean them up, your system can become bloated over time with lots of unnecessary test data. There are two ways we could approach this, and I will show you both of them.

One way is to directly delete the newly created object from the same request. Postman allows you to send API calls directly from a script. You can do this in the **Post-response** section of the **Scripts** tab using the following code:

```
var base_url = pm.environment.get("base_url")
var task_id = jsonData.id;
var token = pm.collectionVariables.get("token")
pm.sendRequest(
```

```

{
  url: `${base_url}/tasks/${task_id}`,
  method: 'DELETE'
},
function (err, response) {
  pm.expect(response.status).to.eql('OK')
});

```

However, if you try to run this, you will see that you get an error saying that this call is unauthorized. We will need to pass in some credentials as follows:

1. On the `{{base_url}}/token` request, go to the **Body** tab and choose the `x-www-form-urlencoded` option and then set the username to `user1` and the password to `12345`.

POST    `{{base_url}}/token`

Params    Authorization    Headers (8)    **Body**    Pre-request Script    Tests    Settings

☐ none   
☐ form-data   
☒ x-www-form-urlencoded   
☐ raw   
☐ binary   
☐ GraphQL

	Key	Value
<input checked="" type="checkbox"/>	username	user1
<input checked="" type="checkbox"/>	password	12345

Figure 11.12: Set the username and password

2. Make sure the request method is POST and send the request.
3. You will get back a token. Copy that token, and go over the collection. On the **Variables** tab, create a variable called `token` and paste in the token value. Save the collection.
4. Go to the Create Task request. On the authorization tab, choose the **Bearer Token** option and then set the value of it to `{{token}}` so that it will use the token you just saved.
5. You can now modify the `pm.SendRequest` to use the bearer as well.

Modify just that part of the script to look like this:

```

pm.sendRequest(
{
  url: `${base_url}/tasks/${task_id}`,

```

```
    method: 'DELETE',
    auth: {
      type: "bearer",
      bearer: [
        {
          key: "token",
          value: `${token}`,
          type: "string"
        }
      ]
    }
  },
  function (err, response) {
    pm.expect(response.status).to.eql('OK')
  });
```

Now if you send the request it should succeed.

## Adding tests to the PUT request

To test the UpdateTask request, I first need to put something in the body of the request so that it will update correctly. I can do so using the **raw** and **JSON** options. I decide to set the body to this value:

```
{
  "description": "modified task",
  "Status": "Draft",
  "created_by": "user1"
}
```

As I am doing this, I realize something. The URL of my request is set to `{{base_url}}/tasks/{{task_id}}` and when I mouse over the `task_id` variable in the URL, I can see that I have set it to 1. When running this request, I am assuming that there will be a task with that ID in the system. For me to be sure that this assumption is correct, I would have to do some setup, perhaps manually. Depending on how things are set up for your application, this might be OK. Perhaps you could prepopulate certain data when the test versions of your app are deployed, for example. However, it is often safest to do the data setup directly with your tests. This gives you the most control over it and the highest degree of certainty that it will be there when you need it. Let's use the **Pre-request** section of the **Scripts** tab to set this up.

I will use the `pm.sendRequest` method. I could just do what I did in the **Create Task** request and manually define the request. However, what I want to do here is create tasks, and as I already have a request defined that creates tasks, let's see if I can reuse it. The first thing I'll need to do is save that request so that I can reuse it. On the **Post-response** section of the **Create Task** request, I will add the following line to the bottom of the file:

```
pm.environment.set("req", pm.request)
```

This will get the current request and save it to a variable called `req`. I can now use this variable to call this request. In the **Update Task** request, I will add the following to the **Pre-Request** section:

```
pm.sendRequest(  
  pm.environment.get("req"),  
  function (err, response) {  
    var jsonResponse = response.json()  
    var task_id = jsonResponse["id"]  
    pm.variables.set("task_id", task_id);  
  });
```

This code gets the request that I saved in the environment and calls it using the `sendRequest` method. It then parses the response and saves the ID in the `task_id` variable so that we can use it in our PUT request. If you now send the request, it will create a task and then modify it.

So, what should we check here? Well, since we know exactly what we specified in the body of this request, we could just check that the response includes that:

```
pm.test("Description matches what was set", function () {  
  var jsonData = pm.response.json();  
  pm.expect(jsonData.description).to.eql("Modified task");  
});
```

## Adding tests to the DELETE request

How do you test a delete request? The first thing to note is that you need something to delete. Just like for the PUT request, we would probably want to first create a task so that we have something to delete. You can do that using the exact same code on the **pre-request** section as we used on the Update Task request.

There also isn't any data to check since it has been deleted. One thing you can check, however, is the response. You can add a test like this:

```
pm.test("Status code is 201", function () {
```

```
pm.response.to.have.status(201);
});
```

We could also double check that the request has been deleted. We could try to get the request and verify that it is not there:

```
var base_url = pm.environment.get("base_url")
var task_id = pm.variables.get("task_id");
pm.sendRequest(
  {
    url: `${base_url}/tasks/${task_id}`,
    method: 'Get'
  },
  function (err, response) {
    pm.expect(response.status).to.eql('Not Found')
  });
```

In this way, we can be sure that the task has actually been deleted.

We have covered a lot in this section, and you have done a lot of work in thinking through this challenge. You've learned how to create automated API tests including a review of some automation test ideas along with how to set up collections and tests in Postman. You then learned how to create several different tests for different kinds of GET requests as well as tests for POST, PUT, and DELETE requests. Along the way, you've learned a lot of tips and tricks for organizing tests, including how to effectively share code between requests and how to use environments. That's a lot of material and this kind of work should be celebrated and shared, so in the next section, I want to show you how you can do just that.

## Sharing your work

If you went through this exercise and built a suite of tests, you will have learned a lot about API test automation. One thing that is important in the modern economy is being able to showcase your work. There are millions of testers and developers working in the industry and you want to be able to stand out from the crowd by showing that you really know your stuff. One great way to do that is by sharing examples of the work you have done and the things you have learned.

I have had a blog for years and have found that to be a helpful way to stand out. I personally love blogging, but it does take time and certainly isn't for everyone. If you want something simpler, there are still ways that you can share your work. You can create a GitHub account, for example, and create public repositories that showcase what you have done and learned.



Within Postman, you can also share what you have done. These features are mostly intended for helping with collaboration between team members, but you can also use them to showcase the work that you have done.

## Sharing a collection in Postman

You can share a specific collection in Postman with the following steps:

1. Go to the collection that you created when doing the previous exercise in this chapter.
2. Choose the **Share collection** option from the menu across the top:

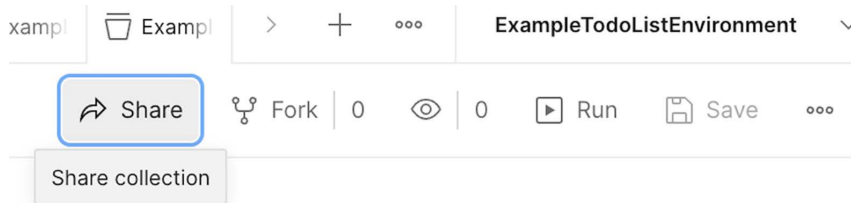


Figure 11.13: Share collection menu option

3. If you were trying to share with teammates, you could just put in their email addresses, but in this case, you are trying to share with the public, so go to the **Via Run in Postman** tab.
4. If you are not in a public workspace, Postman will let you know that you need to move the collection and environment into one. You can use the **Move Collection** button to do that and either select an existing public workspace or create one.
5. You can then click **Next**, copy the code, and use that to embed a **Run in Postman** button on a web page. You can also choose the **Markdown friendly** option if you want to generate Markdown code that you can use in something like a GitHub repository.

If you just want the link, simply go to the **Get public link** tab and get a shareable link from there.

### NOTE:



If you are sharing collections that you made at work, be sure to check if your organization has security policies around sharing data and be sure to follow them. Any time you share data, you should double-check that you are not sharing anything that is private, like internal company data or security keys.

## Summary

This chapter mostly concerned working through an example of creating automated tests for a simple service, but it has taught you a lot of lessons along the way. After learning how to set up the testing site, you learned how to explore the API and find bugs in it. Once you were able to do that, you started to look at how you can automate tests. I gave you a challenge: creating a test automation suite for the test API. I hope you took the time to work through the challenge on your own and were able to learn a lot about what API test automation would look like.

I also walked you through an example of how I would have gone about solving the challenge. While going through my solution, you learned how to set up collections in Postman in a way that organizes the requests so that they will be maintainable. You then further learned about using environments to set up and share variables. I then walked you through creating tests for the different requests.

The tests that I created for the first request were straightforward, although I did show you how to check the properties of multiple items in a list. The tests for the next request built on that and you learned how to set up a test in a folder so that it could be shared across multiple requests. As part of this, you learned how to define a function and save it into an object so that you can call the function from multiple tests.

After that, you learned how to create tests for a POST call. In this section, you learned to use the `sendRequest` method to send an API call from within a test so that you could clean up the test and leave the state of the system unchanged. The tests for the PUT request then showed you how to get some data in a pre-request script that you could then use while cleaning up after the test has been completed. Similarly, we looked at testing DELETE requests and how to set up and verify that they work correctly. The tests that we looked at showed how test design is an iterative process, and you learned how to update and change your plans based on new information that you learned while creating the other tests.

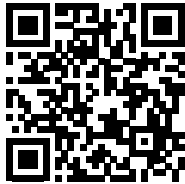
The last thing you learned in this chapter was how to share your work so that you can show others what you have done. You have learned a lot in this book, so don't be scared to share what you have learned with others!

This chapter has taught you a lot about how to create tests for existing APIs. This has helped you learn more about test automation in Postman, but an important tool for automating API tests is being able to mock responses. In the next chapter, we will look at how to use mock servers in Postman.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.com/invite/nEN6EBYPq9>



# 12

## Creating and Using Mock Servers in Postman

Mock servers are a powerful API testing tool. They let you start testing a user interface before the backend is complete. They give you the ability to test applications that use third-party services and enable a lot of other interesting testing scenarios. But what is a mock server, and how do you go about setting one up in Postman? Well, in this chapter, we will dig into those questions and more. We will spend some time learning about mock servers and how to set them up.

This chapter will cover the following topics:

- What is a mock server?
- How to create a mock server in Postman
- How to set up mock data in Postman

### Getting started with mock servers

Mock servers are a powerful tool to have in your API testing toolbox. We will start with a look at what they are, when to use them, and also, importantly, when it doesn't make sense to use them.

#### What is a mock server?

Obviously, in this book, when we talk about a mock server, we talk about a server that mocks an API. In this context, a mock server is simply a server that mocks out the real implementation of the API. In a way, it creates an alternative implementation. Instead of using real API code, we can create our own implementation that returns data. Calling it an implementation is almost saying too much though.

The reality is that most mock API servers do not do much code implementation. Usually, it is just a set of hardcoded responses that are returned when you call certain endpoints, but the point is that you get back those responses rather than the ones computed by the main server.

A mock server can be very simple. It can be as simple as just having a single API endpoint that is hardcoded to return a certain value. It can also be quite complex. You could make a mock server that mocks out all the endpoints in an API and that even has a dynamic element to it. If you really wanted to, you could even create a mock server that is an actual alternative implementation and computes the responses, just in a different way than the real server does.

The complexity of the mock server that you make will depend on what you need to test. Sometimes, you just need to mock out one or two slow requests, so you can make a pretty simple mock server. Other times, you might need to mock out an entire third-party API, so you would need a more complex server.

## When to use a mock server

There are a few different ways to effectively use a mock server. One example is if your app uses a third-party API that you don't want to put under load when you are testing. Rather than sending requests to the third-party service, you could create a mock server that lets you test your application without the need to send those requests to the actual service.

Another situation where it is common to use a mock server is when you need to test stuff on the frontend, but the backend APIs are not yet complete. In that case, you could set the frontend to call the mock server, and this would allow you to develop and test it. This is especially powerful if you have an API specification that dictates what the API will look like. You can then be confident that your mock API does what the real one would, and you can also use the API specification to create the mock server.

Often, we think of UI testing as testing a fully running app through the UI, but it doesn't have to mean this. In fact, it is a good idea to test your UI in isolation from the backend code if the application architecture will let you.

Once again, a mock server can let you do this. You can mock out the backend with something that you run locally, and then test the UI with realistic workflows that run much faster and are more stable than running against a fully running app that sends calls across the network. It also gives you more stability, since the mock backend can isolate you from some of the code changes or bugs on the backend.

Another great use for a mock server is to test scenarios that are difficult to explore otherwise. For example, you might want to check how your frontend responds when the API gives an error, but it can sometimes be tricky to get the backend to produce the exact error that you are interested in. In this case, a mock server can be a powerful tool to let you create the error situations that you want to test.

## Things to be careful of with mock servers

Mock servers are a great tool, but there are a couple of things you should be aware of when using them.

First, don't forget that they don't automatically change in concert with the thing they are mocking. You still need to do some kind of testing with the real service that you are mocking out. You will also probably need to update your mocks on occasion, as the underlying service changes. If you let your mock service drift too far from the real implementation, you can end up not testing the things that you think you are and end up being surprised by things later.

Another thing to watch out for is the accuracy of your mock service. Being a mock, it can't fully reproduce everything about the service it is mocking (otherwise, it would just be a copy of that service), but you need to be careful that it accurately represents the parts that matter. When you populate your mock service with data, make sure the data is representative of what the real service produces.

## Setting up mock servers in Postman

Now that you know a bit about mock servers, let's look at how to set one up in Postman.

To get started, go to the **Mock Servers** option on the left navigation panel:

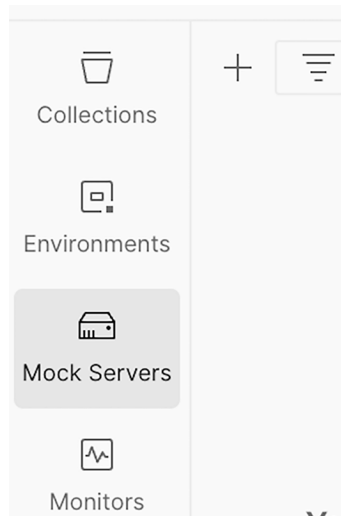


Figure 12.1: Mock Server

If you don't see that option, you might need to configure your workspace sidebar to show it. You can do that by going to the **Configure workspace sidebar** menu option:

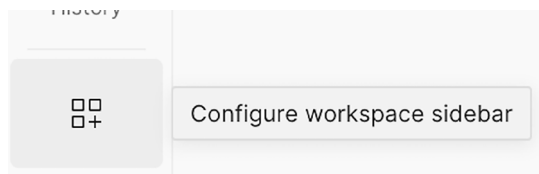


Figure 12.2: Configure workspace sidebar

You can then enable the **mock server** option and any other elements that you want to be visible in your workspace.

Creating a mock server is simple:

1. Click on the **Create Mock Server** option.

You can either create a new collection or select an existing one. For now, just create a new one. In the last chapter, I showed you how to install a todo list application that you can use for testing. Let's look at mocking one of its endpoints.

2. Enter **tasks** for the path that you want to mock. Leave the **Response Code** at 200, and set the response body to `[]`.

This says that we want the tasks endpoint to return an empty list. This is the kind of thing that can be a bit tricky to test without a mock server, as you might not have a test site that has no data on it. However, it is also an important thing to check, since new users will have an empty database when they get started.

3. Click **Next** and configure the server. Give it a name – something like **Test Mock Server**. You can leave the rest of the settings at their defaults, and then click on **Create Mock Server**.

It's as easy as that to create a mock server. In order to start using it, all you need to do is copy the mock server URL and use it in whatever context makes sense. You could paste it into your web browser's address bar. If you paste in the **URL**, add `/tasks` to the end of it, and hit *Enter*, you will see the blank list that you specified:

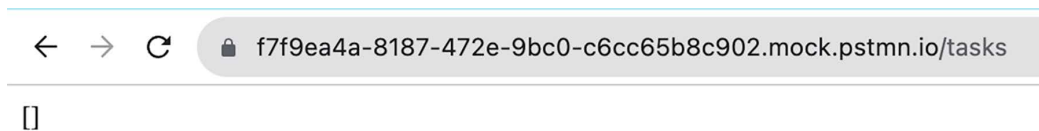


Figure 12.3: Blank list from the mock server

## Modifying mock server values

Now that you have the mock server up and running and you can send requests to it, let's look at managing its data.

When the server was created, Postman automatically created a new collection for it that has the same name as your mock server. You can find it by going to the **Collections** tab, and if you open it, you will see one request in it. This request is the **tasks** request that you specified when making the mock server. If you expand it in the navigation tree, you can see that it has an example under it:

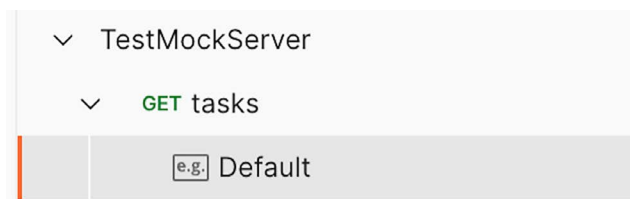


Figure 12.4: Mock example



This example is what defines the response that the mock endpoint gives back. In the **Body** field of the response, you can modify the response. Set it to something like this:

```
[
  {
    "description": "Mock Task 1",
    "status": "Draft"
  }
]
```

Save the request, and then go to the web browser tab that you previously called the mock from and refresh it. You should now see that it has the new values you put in. So that is how you modify the values that a mock endpoint returns, but what about if you want to add additional values?

## Creating more mock values

When dealing with different scenarios, you might want your mocked endpoint to return different values. You can set up an endpoint to return different values, by adding additional examples to the mocked request:

1. Select the **Add example** option from the request menu:

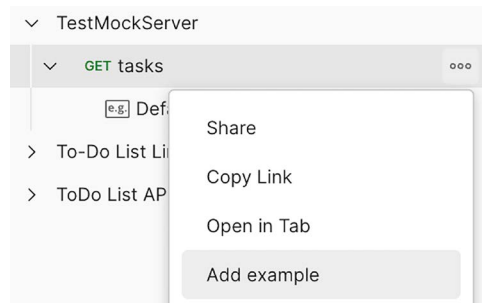


Figure 12.5: Add a mock example

2. Call this example **500Error**.
3. Set the **Status code** to **500** and the return value to **Server Error**, and then click save:

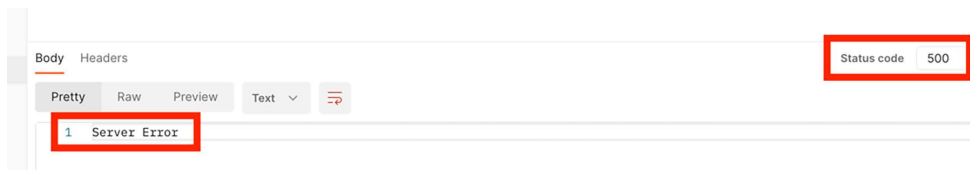


Figure 12.6: Add an error response

This example is now ready for the mock server to send to us, but this same endpoint now has different possible responses. How do we tell the server which response we want it to return to us? Postman has a set of rules that it uses to pick which example to return when you send a request to the mock server. It will look at things like the URL path you use, the method you send it with, and any query or body values, but in this case, all of those things match.

In a situation like this where everything matches, Postman gives you the ability to more explicitly specify which example you want to use. It provides a few different custom headers that you can use to specify which example you want back. One of them is the **x-mock-response-code** header. This header tells Postman to pick the example based on the response code. In this case, we could set that header to 500 to get back the error response and 200 to get back the normal response. Let's try this out:

1. Create a new request in Postman that uses the mock server and points to the /task endpoint.
2. On the **Headers** tab, create a new header called x-mock-response-code and set its value to 500.
3. Send the request, and you should get back the 500 response:

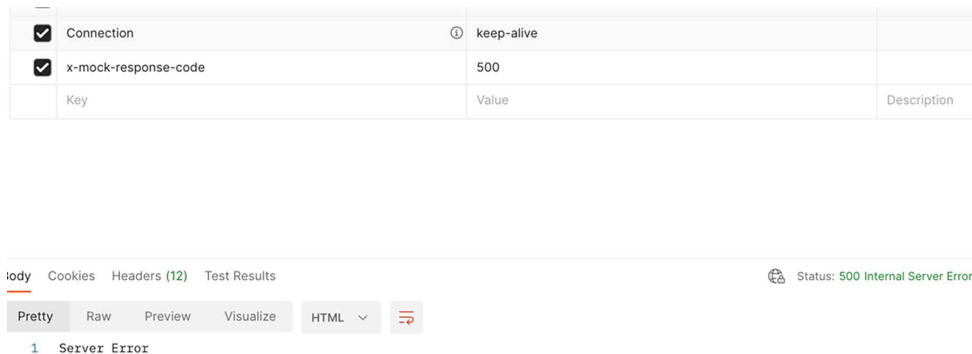


Figure 12.7: Error response

4. Change the header value to 200 and send the request again. You should now get back the data you specified in your 200 example.

## Mocking route parameters

What do you do if you want to mock an API path that has parameters in it? For example, in the todo list application, you might want to mock the route /tasks/:id, where the ID can be any positive integer. How do you mock something like this? Well, let's try it out:

1. In the TestMockServer collection, create a new request called task.

2. Set the **URL** to `{{url}}/tasks/{{task_id}}`.
3. Save the request.
4. Add an example to the request called `Default`.
5. Set the status code to `200 OK`.
6. Set the response body to something like this:

```
{
  "description": "Mock Task {{task_id}}",
  "status": "Draft"
}
```

Now, if you send a request to `/tasks/1` using the mock server, you will get back a response with the description `Mock Task 1`.

## Mocking dynamic data

Mocking GET calls isn't too bad, but it can get a bit harder when you want more dynamic data, as you might want with a POST or a PUT call. When you call an endpoint with the POST method, you want it to return the object that you sent it. As you might guess from the example above, this can be done in Postman by using variables:

1. In the `TestMockServer` collection, create a new request called `Create Task`.
2. Set the method to POST and the **URL** to `{{url}}/tasks` and save it.
3. Add an example to the request called `Create Task`.
4. Set the status code to `201 Created`.

In order to return the body of the request that is sent, you can use the `$body` variable in the response body of the example. This variable contains the body data that is sent to the mock server, and you can use it to echo back the data that is sent. The syntax to use it is a bit tricky though. To send back the data, create a response in the example that looks like this:

```
{
  "description": {{$body 'description'}},
  "status": {{$body 'status'}}
}
```

There are a couple of things to note about this. First, the `$body` variable needs to be inside of the double curly braces. If you don't wrap it in the curly braces, the response will just return `$body` instead of echoing back the data that you sent.

The second thing to notice is how you access the different attributes of the body. In order to access them, you need to include them in a string inside the same set of curly braces that the `$body` variable is included in. If you have nested attributes, you can access those by using dot notation. Something like `description.subattribute` would access a sub-attribute that was under `description` in the body.

So now, let's try this out:

1. Create a new request in Postman (that is not in the Test Mock Server collection).
2. Set the request **URL** to the URL of the mock server with `/tasks` at the end of it.
3. Set the method to **POST**.
4. On the **Body** tab, change the type to raw and choose **JSON** from the dropdown.
5. Put in a body that looks like this:

```
{
  "description": "This is a mock task I am creating",
  "status": "Draft"
}
```

6. Send the request, and the mock server should echo back to you the body that you sent it.

If you wanted to mock a PUT request, you might not know which part of the request will be modified. In that case, you can use default attributes:

1. Duplicate the Get Task request in the Test Mock Server collection.
2. Rename it update task and change the method to PUT.
3. Save the request.
4. Update the example for this request to also use the PUT method.
5. Set the example body response to something like this and save it:

```
{
  "description": {{$body 'description' 'Mock Task {{task_id}}'}},
  "status": {{$body 'status' 'Draft'}}
}
```

In this case, the second statement in quotes after the `$body` will be used as a default if you don't specify that field. For example, try calling the mock server with the following steps:

1. Create a request and set it to use a PUT method.

2. Set the **URL** to `{{url}}/tasks/10`, where the URL variable holds the value of the URL for your mock server.
3. Set the body of the request to this:

```
{  
  "status": "Approved"  
}
```

Send the request, and you will notice that you get back a response like this:

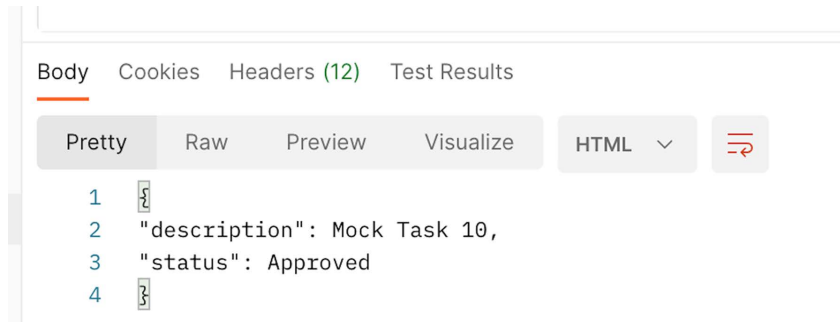


Figure 12.8: Mock response with a default description

Notice that although you did not specify the description in the original request, the response has loaded the default and sent that back. Alternatively, you could specify the description and not the status when you sent the request, by making the body look like this:

```
{  
  "description": "Changed the task description"  
}
```

In this case, you will get back a response that looks like this instead:

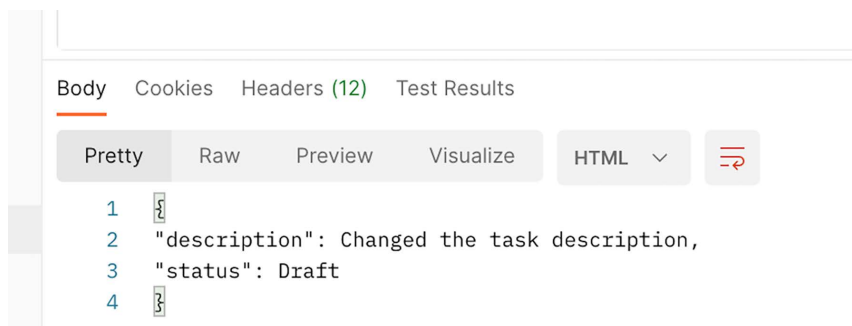


Figure 12.9: Mock response with default status

You can see that it now uses the description that you specified, but for the status, it returns the default status specified in the example.

## Using mock servers

I've already touched on some of the times when you might want to use mock servers. You might want to use them to mock out third-party dependencies, enable or improve your testing, or even allow you to develop a part of your product before the API is completed. But how do you actually go about doing this?

One thing that you will need to do is to share your mock server. It's great that you can create and use a mock server on your computer, but how do you allow others to also use it? The good news is that as long as you did not select the option to make your mock server private when you created it, other people can already use it. Postman has created a public URL for your server!

If you did make it private and you want to change it (or if you want to make a public server private), you can do so by going to the mock server and then choosing the **Edit Configuration** button:

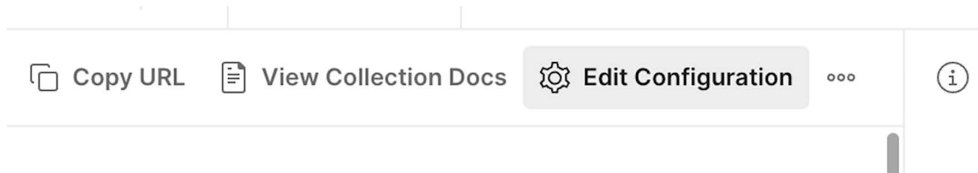


Figure 12.10: Editing the mock server configuration

Then, deselect the **Make mock server private** option to make the server public.

## Using private servers

Public mock servers are a bit easier to work with, since you don't need to specify any authorization to use them, but you might want to use a private server if you work with intellectual property that cannot be shared publicly. I would generally advise that you try to keep any private or personally identifiable information out of your mock servers, but sometimes, it is difficult to set up realistic testing scenarios without using that type of information. In that case, you would want to use a private mock server.

If you did have a private mock server, you need to have some way of letting Postman know who is allowed to use it. Postman will use the `x-api-key` header to do this. In order to get an API key, you can use the following steps:

1. Go to <https://go.postman.co/settings/me/api-keys>.

2. Click on the **Generate API Key** button.
3. Name your key something like `ForPrivateMockServer` and generate the API key.
4. Copy the API key and store it somewhere safe (like in a password manager). You can use this key to do other things in Postman as well, so it can be worth keeping track of it for later.

Now, in order to call a private mock server that you have created, all you need to do is add an `x-api-key` header to your request to the mock server and specify the value of it as your API key. Note that you will want to make sure that your API key stays private, so be sure to store it in a secure variable in Postman.

## Mocking a third-party API

A good use of mocking is to mock out third-party APIs. This can be helpful for testing purposes, as it allows you to test the core functionality of your application without needing to have access to (or put a heavy load on) any third-party service that your app might depend on. For example, let's imagine that you are testing an e-commerce application that integrates with a **customer relationship management (CRM)** system. Your application queries the CRM to get some information about the customers, but when you are testing, you don't want to use real customer data due to privacy concerns. In that case, you could set up a mock API that you could use instead that would return some fake data. Let's look at how you might do that.

Let's say that the API endpoint to get customer information from the third-party CRM looks something like this: `{{url}}/crm/users/{{user_id}}`. When you call it, you get back a response that looks like this:

```
{
  "country": "US",
  "city": "New York",
  "language": "en_US",
  "street": "123 Main Street",
  "Currency": "USD",
  "id": "738964000000291009",
  "state": "New York",
  "first_name": "Dave",
  "email": "dave@dave.com",
  "zip": 12345,
  "created_time": "2023-10-26T13:00:28+00:00",
  "last_name": "Westerveld",
  "time_zone": "GMT",
```

```
    "phone": "9597538144",
    "dob": "2003-10-26T13:00:28+00:00",
    "status": "active"
  }
```

If the data in here was really user data, you might run afoul of privacy laws, so let's see if we can create some fake data for this that we can use in a mock server. For now, we will just use the mock server we have already created:

1. Create a new request in the mock collection and call it User.
2. Set the **URL** to `{{url}}/crm/users/{{user_id}}`:
3. Create an example under the request using the same URL, and set the body to look like this:

```
{
  "country": "US",
  "city": {{$randomCity}},
  "language": "en_US",
  "street": {{$randomStreetAddress}},
  "Currency": "USD",
  "id": "738964000000291009",
  "state": "New York",
  "first_name": {{$randomFirstName}},
  "email": {{$randomEmail}},
  "zip": 12345,
  "created_time": {{$randomDateRecent}},
  "last_name": {{$randomLastName}},
  "time_zone": "GMT",
  "phone": {{$randomPhoneNumber}},
  "dob": {{$randomDatePast}},
  "status": "active"
}
```

4. In another request in Postman, call the mock server with the `/crm/users/1` appended to the end of the URL. You should get back a response with the different values filled in.

You can see from this example that Postman provides a lot of different methods to help you fake out different kinds of data. You can see a full list of the different kinds of random data that Postman can generate in the documentation here: <https://learning.postman.com/docs/writing-scripts/script-references/variables-list/>.



## Summary

This chapter has introduced you to the concept of using mock servers in Postman. You have seen how to set up a mock server in Postman. You have learned about making the server private or public. You have also learned a lot about how to set up data for your mock servers.

In this chapter, I've shown you how to create very simple mocks that respond with a static response every time they are called. I've shown you how to do more complex mocking where you can respond with more dynamic values. We also got into the details of how to respond with mock data that is a mix of dynamic and static data, and that has default responses if you don't specify the full body of the request.

Mocks can be used for a variety of different purposes, and we explored some of the reasons you might use them and some things to be careful of when you work with them. We dove into some examples of using mocks for things like testing with third-party services, and we looked at how to generate random yet realistic data when you need to have anonymized data.

Mocking is a powerful tool in both the tester's and the developer's toolbox, and Postman provides a lot of useful tools to start using it. Mock servers can be used as a type of contract for how a server is expected to work. In the next chapter, we will look at contract testing in depth and see how it can help you with API development.

## Leave a review!

Enjoying this book? Help readers like you by leaving an Amazon review. Scan the QR code below to get a free eBook of your choice.



# 13

## Using Contract Testing to Verify an API

We've all had to sign a contract before. Maybe it was when you were buying a house, starting a new job, or opening a bank account. There are many reasons that we sign contracts. I have a filing cabinet where I keep important paperwork and I suspect that about half of the documents in there are contracts of some sort.

Not all contracts are explicit. If I wanted a load of gravel for my driveway, I could call up a gravel company and ask them to deliver it. They would do so under the implicit contract assumption that I would pay them for the gravel once they'd delivered it and given me an invoice. Implicit contracts like this can work well but they are more susceptible to errors or misunderstandings. Some implicit contracts are based on cultural understanding, and there may be times when things get confused or forgotten. In general, the more important something is, the more there is a need for an explicit instead of implicit contract. You will see formal contracts for bigger and longer-term things such as buying a house or starting a new job, while implicit contracts will suffice for many of the ordinary things of life.

But what does all this have to do with API testing? Well, one powerful API testing concept is called **contract testing**. In contract testing, we create a document that works as a contract that we can test against. Much like the contracts we see in everyday life, they set out the things that each party must do. In the case of API testing, the two parties involved in the contract are two different pieces of code. An example would be the frontend and the backend of a system or two services that communicate with each other. The contract establishes the way that those two parts of the system will communicate with each other.

Often, software is built without an explicit contract between the different pieces of the code. Developers will make assumptions about what the different pieces of code can do. An implicit contract like this can work well when the code is small or relatively unimportant but it is often helpful to have an explicit contract between different parts of your code.

In this chapter, you will learn all about what contract testing is, how to set up contract tests in Postman, and how to create shared contract tests that can be run and fixed by all the affected teams. We will cover the following topics in this chapter:

- Understanding contract testing
- Setting up contract tests in Postman
- Running and fixing contract tests

## Understanding contract testing

In this chapter, we will learn how to set up and use contract tests in Postman, but before we do that, it's important to make sure that you understand what they are and why you would use them. So, in this section, we will learn what contract testing is. We will also learn how to use contract testing and then discuss approaches to contract testing – that is, both consumer-driven and provider-driven contracts. To kick all this off, we are going to need to know what contract testing is. So, let's dive into that.

### What is contract testing?

I have already talked about what contract testing means at a basic level. Contract testing is a way to make sure that two different software services can communicate with each other. Often, contracts are made between a client and a server. This is the typical place where an API sits, and in many ways, an API is a contract. It specifies the rules that the client must follow in order to use the underlying service. As I've mentioned already, contracts help make things run more smoothly. It's one of the reasons we use APIs. We can expose data in a consistent way that we have contractually bound ourselves to. By doing this, we don't need to deal with each user of our API on an individual basis and everyone gets a consistent experience.

However, one of the issues with an API being a contract is that we must change things. APIs will usually change and evolve over time, but if the API is the contract, you need to make sure that you are holding up your end of the contract. Users of your API will come to rely on it working in the way that you say it will, so you need to check that it continues to do so.

When I bought my home, I took the contract to a lawyer to have them check it over and make sure that everything was OK and that there would be no surprises. In a somewhat similar way, an API should have some checks to ensure that there are no surprises. We call these kinds of checks contract testing. An API is a contract, and contract testing is how we ensure that the contract is valid, but how exactly do you do that?

## How to use contract testing

We will learn how to create and run contract tests in Postman shortly, but first, let's look at how the process works. There are a couple of different ways that you could approach contract testing. One possible way is to create a collection of tests that exercise the API in all its configurations. You would then run that set of tests every time a change was made to the API to ensure that nothing has changed. Technically, when you are doing this, you are checking the contract, but this is not what we would call contract testing.

With contract testing, you want to check just the contract itself. You don't want to run a full regression suite. Part of the value of contract testing is that it allows you to just verify that the contract is correct without needing to include full API calls. However, in order to do that, you need to be able to check the contract somehow. How do you do that?

The best way to do this is by having the contract documented somewhere. This is usually done with some kind of specification. In the previous couple of chapters, I've been showing you how to work with an OpenAPI Specification file. If you have a specification that defines all the actions that the API can perform, it will work well as the basis for your contract tests. In fact, in the previous chapter, you learned how to do one half of a contract test.

Mock servers are, in essence, contracts. They provide the frontend, or **API consumer**, with a contract that they can use. However, as we saw in the previous chapter, there are some limitations to this. Although the mock server tells the consumer what they can do, it doesn't get used when you are working on the backend code. This means that the API might change without the API consumer knowing about it. This is because those who are working on the backend code don't know what things the users or their API are relying on. They could read through various examples in the mock server and try and figure it out, but that would be a time-consuming and low-value process.

What contract testing does is create a set of tests that the API producer can run to verify that any changes they've made are not breaking any of the consumer's needs. These contract tests give the API producer something they can run, without needing to worry about the details of the consumer implementations. Well, this has been a lot of words and might seem a bit abstract, so let's take a look at a more concrete example of what this might look like.

A typical test for an endpoint might look kind of like this:

```
pm.test("Check the name", function () {  
    var jsonData = pm.response.json();  
    pm.expect(jsonData.name).to.eql("Bob");  
});
```

In this case, we are checking that the response gave back the expected value of “Bob” for the name, but in a contract test, we aren’t worried about what specific values we get back – we understand that people will use this API for many different names – rather, we are concerned with checking that we have the right kind of data. So, rather than checking if the name is “Bob”, we would check things like ensuring that the data is a string and doesn’t have any numbers. Or we might check that the name field is a top-level object and not somewhere else in the response.

We will get into more concrete examples of contract testing soon, but before we do, we need to discuss who gets to define the contracts. There are always at least two parties involved in a contract, and with APIs, those two parties are usually known as the **provider** and the **consumer**. Which one of them gets to define the contract? Well, let’s look at that next.

## Who creates the contracts?

Before we get into the details of setting this all up in Postman, there is one more thing we need to discuss. Who creates these contracts? Should the consumer of the API be the one creating the contract, or should it be the provider? Much like a real-life contract, there is probably going to be some need for negotiation and the ability to reject a contract proposal, but somebody has to put the contract forward in the first place. There are two main approaches to this: you can either have consumer-driven contracts or you can have provider-driven contracts. Let’s look at each one in more detail.

### Consumer-driven contracts

**Consumer-driven contracts** are contracts that the consumer makes. In this case, the consumer defines what their needs are from the API and provides contract tests that show the ways in which they will be using this API. These can be defined directly up-front in a design-driven manner, where the consumer needs to define the details of how the API gets designed in the first place. Alternatively, they can be provided for existing APIs by showing them which part of it this particular consumer relies on.

There are several advantages to consumer-driven contracts. The whole point of an API is to provide value to those who consume it. By letting the consumers define what the API does, you are helping to ensure that you are delivering information that is valuable to the users of your API. Another advantage is that it can help the API provider understand the different ways that consumers are using the API.

There is a “law” known as **Hyrum’s Law**, which observes the following:

---

*“With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.”*

– <https://www.hyrumslaw.com/>

---

This means that your system will get used in ways that you did not anticipate it would when you were designing it. However, if the users provide you with contracts that show you how they are using it, you can learn from that and not make changes that break workflows in unintended ways.

There are some downsides to this, though. If you have a lot of people consuming your API, you will have to run contract tests for each of them every time you make changes. There will probably be a lot of overlap between these tests, which leads to redundancy and inefficiency in your testing. Another concern is that, sometimes, the consumers tend to do some unexpected things that can really constrain future development. For example, I recently saw a case where a client was using an API that contained some information about another domain in it. The client wanted this information, so they got it from the API we had provided. However, we wanted to change the API so that it no longer returned the data and, instead, returned it from the correct domain. The problem was that the client had come to depend on the data being in the “wrong” spot, which made it much harder for us to change the API to work the way we wanted it to. If you have some feature in your API that you put there because of a consumer contract, it might be a lot harder to change in the future. The consumer can point to the contract they provided and point out that you had earlier agreed to do things this way. You don’t want to be in a situation where one client’s use case prevents you from creating useful value for others.

## Provider-driven contracts

Instead of having the contract primarily driven by the consumers, you could have the API producer be the one providing the contract. In this case, the team creating the API would create a contract that defines what data that API provides and what format that data will be in when clients request it. Consumers could then use that contract to build mock servers that they could use for testing and development.

This approach has some benefits. First, it allows API providers to set out a standard set of actions and data that are supported. According to Hyrum's Law, users will still end up using things in ways that are not defined by the contract, but then at least the API provider can explicitly say that those are not the supported ways to use this API, and they don't have to worry too much about breaking those workflows. Another benefit of this is that it is much more scalable. It doesn't really matter if you have two users or two million users. You have the same number of contract tests to run as the API provider.

The biggest downside of provider-driven contracts is the missing feedback loop. In this case, it is much harder for API providers to discover the interesting or unusual ways in which clients are using their APIs. This also means that since those configurations are less understood, they will be easier to break. Even if the provider can point to the contract and let the consumer know that they were using unsupported functionality, it is still never nice to break things that your customers depend on. You can use things like telemetry tools to help you understand this, but as with everything in software development, things come with trade-offs. Generally, I would recommend provider-driven contracts, but it is good to be aware of the trade-offs you are making when you take this approach.

Now that you have a good grasp of what contract testing is and how it works, let's look at how to set up contract tests in Postman.

## **Setting up contract tests in Postman**

Creating contract tests is similar to creating any other API test: you need to create a collection and then make requests in that collection with tests in them. But what kind of tests do you create when you perform contract testing?

The idea of contract tests is to describe all the different parts of the API that you need, but not to do things such as look at how to use them in a workflow or other aspects of testing. Contract tests are a good way to set the minimum bar for what needs to be working, but you will need to do additional testing beyond them for it to be effective. Of course, as with any other testing that's done in Postman, you will need to create a collection for your contract tests.

## Creating a contract testing collection

Any collection can be used for contract testing, but if you start from an API specification, Postman provides some nice shortcuts for you. In *Chapter 3*, we made an OpenAPI specification file and imported it into Postman. If you have not yet worked through that content, you can use the file provided in the Chapter13 folder of the GitHub repository for this course (<https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/master/Chapter13>). If you already set this up in *Chapter 3*, you can skip the following steps but, otherwise, you will need to import the OpenAPI Specification file using the following steps:

1. Choose the **Import** button at the top left of the application.
2. In the file browser, navigate to where you saved the `budgeting.yaml` specification file and click **Open**.
3. On the resulting dialog, select the **OpenAPI with a Postman Collection** option and then click on **View Import Settings**.
4. Ensure that the **Parameter generation** option is set to **Schema** and then go back to the import dialog.
5. Click on **Import**. This will automatically import the API and create a collection.
6. Once the import has completed, go to the API section in the navigation tree, expand **Budgeting API** and then **Definition**, and click on `budgeting.yaml` to see the **raw** definition file that you just imported.

You will also need a mock server to make requests against. Once again, you may have already created this in *Chapter 3*, but if not, you can go to the **Collections** tab and click on the ellipsis menu beside the **Budgeting API** collection. Choose the **Mock collection** option and Postman should create a mock server for you.

Now that you have a collection associated with the open API specification, let's look at what happens if you change the specification:

1. Click on **budgeting.yaml** in the API navigation tree.
2. Change the first path from `items` to `changedItems` and save the file.
3. Click on the API at the top level of the navigation tree.



4. Under the **Collections** section, click on the ellipsis menu and choose **Enable update suggestions from definition**.

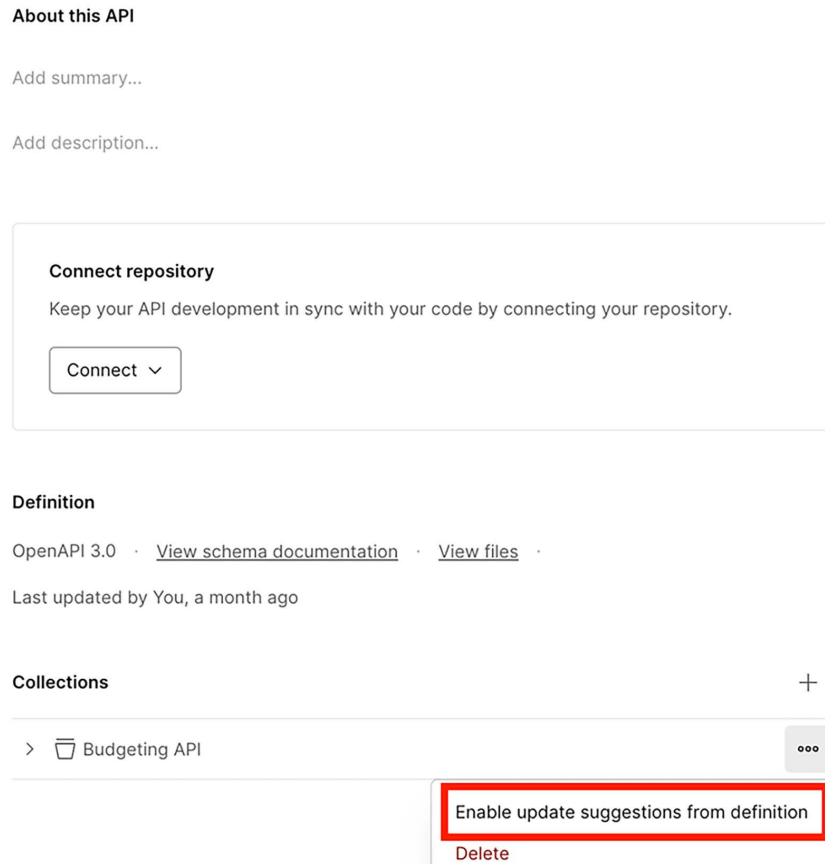


Figure 13.1: Enabling update suggestions

5. Now go to the **Budgeting API** collection on the APIs tab and you should see a refresh icon indicating that the collection can be updated in response to the change made in the specification.

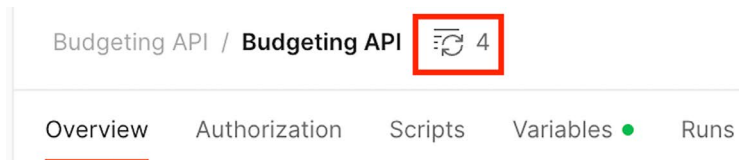


Figure 13.2: Refresh icon

6. Click on this button to preview the changes that are going to be applied.
7. If you wanted to, you could update the collection. If you do this, make sure to select the checkboxes beside the requests in the **Remove requests** section and then click the **Update Collection** button to update the collection.
8. You can then change the specification back to what it was (so change the path back to `/items`) and refresh and update the collection so that it is back to where it was.

This makes it easy to keep the collection and the API definition in sync with each other, but Postman will also validate things from the other direction as well. It will check any requests that you create in a collection that are associated with a specification to ensure that they are properly conforming to the specification. You can see how this works with the following steps:

1. Click on the **Successful Operation** example under the **Get the list of budget line items** request.
2. This example should have a couple of example transactions. Change the amount field of one of them to have a value of `wrong` instead of a number.
3. Save the changes.
4. When you do this, you should see little orange dots appear, and a note that there is an issue. These indicate that you have an example that does not match up with what the specification expects.

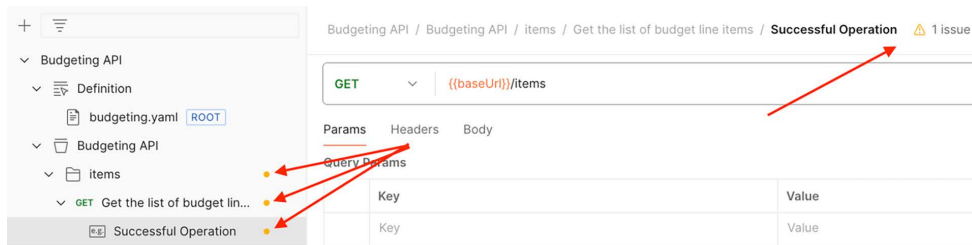


Figure 13.3: Wrong example

5. If you click on the **1 Issue** note it will tell you what the issue is.
6. Change the value back to a number and save the example again, and the error notifications should go away.

You can see that linking a collection with an API specification gives you a lot of built-in validation that helps you ensure that you are not violating the contract.

You can also update your specification if necessary. For example, if you change the **amount** field in one of the transactions in the **Successful Operation** example to something like 12.1, you will notice that you get an error. This is because, in the specification, we have said that this field needs to be an integer.

However, we now realize that it should be a number type instead since amounts can be decimals. We can update it:

1. Click on **budgeting.yaml** in the navigation tree.
2. Scroll down to the **schemas** section at the bottom of the file and, under **properties**, find the **amount** section.
3. Change the type from **integer** to **number** and save the definition file.

If you now look at the example, there should no longer be any issues reported when you are using decimal numbers for the amounts.

## Adding tests to a contract test collection

You already have a lot of built-in validation, but you aren't quite done yet. Postman has automatically created the structure and details of the request, but you still need to define the actual tests that you want this collection to run. This is the point at which understanding the theory of contract testing becomes helpful. What kind of tests should you add to this collection?

With contract tests, we want to make sure that the API is fulfilling the needs that we have as a consumer. We don't want to create a comprehensive set of tests that checks everything this API can do. We just want to add tests that verify that it provides the data we are interested in. In this case, we are pretending that we have a budgeting application, so we would only want to add tests that verify data for the kinds of requests that our application is running. We would then create tests that check that those specific calls give us back the data that we need.

We could manually create some tests like this, but Postman has a nice validation collection that they've made, which we can copy into our project. You can do that with the following steps:

1. Go to the search bar at the top of Postman and search for **contract test generator**.

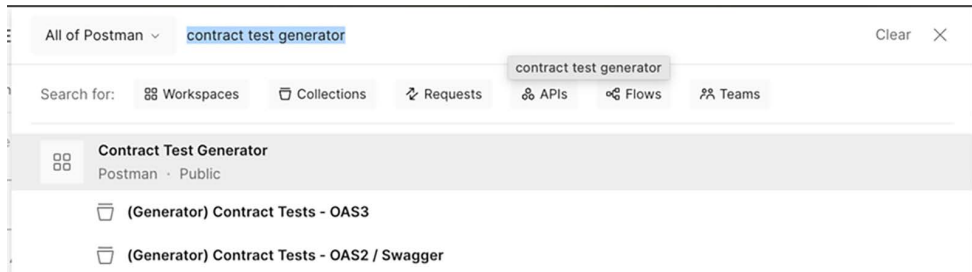


Figure 13.4: Searching for contract test generator

2. Select the **(Generator) Contract Tests – OAS3** collection.
3. On the menu beside the **(Generator) Contract Tests – OAS3** collection, select the **Create a fork** option.

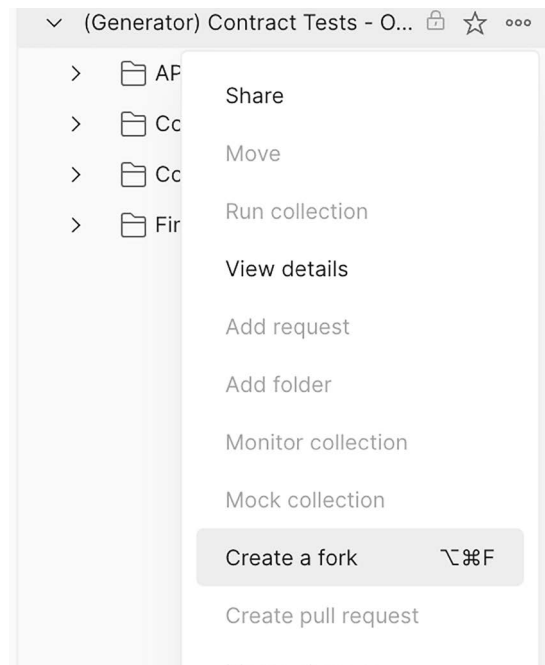


Figure 13.5: Creating a fork

4. Name your fork, and make sure you have selected the workspace that you are currently using as the destination in which the copy of the collection will be created.

5. Click on **Fork Collection**.
6. If necessary, choose to make your profile public and, after a moment, you should see a forked copy of the collection appear in your workspace.
7. Use the search in Postman to return to the **Contract Test Generator** workspace.
8. Go to the **Environments** tab and on the menu beside the **Contract Test Environment**, choose the **Create a fork** option.
9. Name the fork that you are creating and, once again, make sure that you have the correct workspace selected to copy it into and click on **Fork Collection**.

Find the env-server variable in the environment that you just copied. This variable needs to be set to the server URL that you have specified in your OpenAPI Specification file. If you look in the file, you can see that this is `http://localhost:5000/budgeting/api`.

10. Set the **Current value** of the env-server variable to `http://localhost:5000/budgeting/api`.

You will also need a **Postman API key**.

11. Click on your profile option at the top right and choose the **Settings** option.
12. This will take you to your account page in the web browser. Choose the **API keys** option.
13. If you don't yet have an API key, you can create one using the **Generate API Key** button. Otherwise, you can use an API key that you have already generated.
14. Back in Postman, find the env-apiKey variable in the **Contract Test Environment**, set the type of it to **secret**, and paste in the API key.

One last thing we will need to set this up is a **workspace key**.

15. In order to find the workspace key, click on the name of your workspace on the top-left side of Postman.
16. On the far-right side, click on the ellipsis menu and choose the **Workspace info** option.

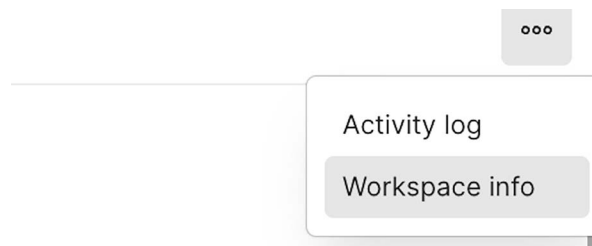


Figure 13.6: Workspace info settings

17. From there, you can copy the workspace ID and paste it into the **Current value** field of the `env-workspaceId` variable in the environment.
18. Make sure to save the environment and make sure that it is set as the active environment.

## Running contract tests

Now you are ready to run this collection. Go to the **(Generator) Contract Tests – OAS3** collection and run it. This collection will verify that the schema is set up correctly. When you run it, you will see some failures.

### GET Verify Component Adherence

<https://postman-echo.com/delay/0>

```

FAIL   Schema 'item' begins with an uppercase letter | AssertionError: expected 'i' to equal 'I'
PASS   Schema 'item' has required property 'transaction_date' defined
PASS   Schema 'item' has required property 'amount' defined
PASS   Schema 'item' has required property 'category' defined
PASS   Schema property 'item.transaction_date' is lowercase
FAIL   Schema property 'item.transaction_date' has a description between 10 and 100 characters | A:
FAIL   Schema property 'item.transaction_date' has an example | AssertionError: expected { type: 'str
PASS   Schema property 'item.amount' is lowercase
FAIL   Schema property 'item.amount' has a description between 10 and 100 characters | AssertionEi
FAIL   Schema property 'item.amount' has an example | AssertionError: expected { type: 'number' } tc
PASS   Schema property 'item.category' is lowercase
FAIL   Schema property 'item.category' has a description between 10 and 100 characters | Assertion
FAIL   Schema property 'item.category' has an example | AssertionError: expected { type: 'string' } to
FAIL   Schema 'itemId' begins with an uppercase letter | AssertionError: expected 'i' to equal 'I'

```

Figure 13.7: Contract test failures

One of the things these tests check is that you are using best practices in your schemas. You can see that the first error expects that schema property names will begin with capital letters. You can update the property names in the **schemas** section of the definition to start with capital letters:

1. Go to the APIs section and click on **budgeting.yaml**.
2. Scroll down to the components section at the bottom of the file and, in the **schemas** section, update `item`, `itemId`, and `items` so they all start with capital letters.

3. Once you have done that, you will also need to update the paths and components where you reference those schemas. These will be spots that look like this:

```
$ref: "#/components/schemas/item"
```

You will need to update the `item` at the end to be `Item`. You will also need to do the same things for the references to `items` and `itemId`.

4. Once you have found and updated all references, be sure to save the specification.

There are also some errors about how the schema properties need to have descriptions and examples, so let's add them:

```
components:
  schemas:
    Item:
      type: object
      required:
        - transaction_date
        - amount
        - category
      properties:
        transaction_date:
          description: "date transaction occurred"
          type: string
          format: date
          example: 10/10/2024
        amount:
          description: "amount of the transaction"
          type: number
          example: 12.25
        category:
          description: "category of the expense"
          type: string
          example: "Groceries"
```

Figure 13.8: Adding examples and descriptions

After you have updated the schema, you can run the collection again and you should see all the tests passing.

As you can see, this collection provides a lot of powerful checking. In the background, it is creating a lot of tests and running them. It doesn't save those tests, but you can rerun this collection any time you want, and it will recreate and rerun those tests based on the latest changes to your specification. This is a great way to make sure that what you are doing in your collection matches up with what you have defined in your specification. It also helps to ensure that your schema is properly defined and follows all the rules of the OpenAPI schema.

It is important to have this if you want to use your OpenAPI specification for other things like generating code.

## Using Postman Interceptor

When creating contract tests, you want to create tests that demonstrate the actual needs of your application. Postman has a tool called Interceptor that allows you to do this. When it is running, it will capture all the requests that your application makes, allowing you to see the actual requests that you need for your contract test. You can set up Interceptor in Postman by following these steps:

1. First, you will need to have Chrome installed on your computer. I would also recommend that you close all tabs except one so that you don't get too much unwanted data. You can then install Postman Interceptor by downloading it from the Chrome web store. You can download it directly from this link: <https://go.pstmn.io/interceptor-download>.
2. Click on **Add to Chrome** and choose **Add extension** on the verification prompt to install the extension.
3. Once the extension has been installed, click on the extension to open it.
4. At the bottom of the extension flyout, type a url (you can use localhost if you have a local server running) into the URL filter and hit *Enter*.

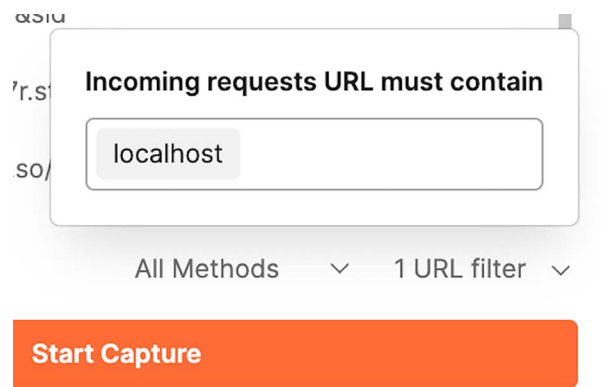


Figure 13.9: Filter Postman Interceptor



5. Click the **Start Capture** button.
6. In your web browser address bar type in the URL of the site that you set the **URL filter** for.
7. Open the Interceptor extension again and click the **Stop Capture** button.
8. Chrome will prompt you to open the Postman app. Select to do so.

Postman will open with the requests loaded. You can see the details of a request by clicking on that row in the table and Postman will show you the details of what data was sent and received with that request.

You can then save the requests to a collection:

1. Click on the checkbox beside all the requests that you want to save.
2. Click on **Save Requests**.
3. At the bottom of the popup, choose the **New collection** link to make a new collection.
4. Name that collection `InterceptedRequests` and create it.
5. Choose to organize the requests by **Endpoints** and save.

You can now go to the collection and see that all the requests have been added along with any data that had been set up for them. As you can see, this allows you to easily add the requests that your application sends to your contract testing collection. You will still need to create tests to check that the data and data structures are correct, but this can help you figure out what requests to send for contract tests.

I'm not going to walk you through a step-by-step process for creating tests for these requests, but now that you know what requests to include and what kinds of parameters they need to have, you should be able to go through this and fill in the test's details. However, in order to understand this a bit better, let's take a look at what you would do in broad terms.

You would want to use Interceptor in situations where you do not have an already defined schema for the API that you can create contract tests against. In that situation, you can use Interceptor to help you "reverse-engineer" how the API works and what things it provides. So, in this example, we could look at the results given back by a couple of different actions in the user interface and use that information to figure out what settings the API needs.

Using that information, we could build out a schema that describes the rules for these fields and then use that to create contract tests. However, even after that is done, you will still need to ensure that these tests are available for everyone who has agreed to use this contract. Everyone involved needs to be able to run and fix these tests; otherwise, they are just ordinary API tests.

## Running and fixing contract tests

In many ways, contract tests aren't that different from other tests that you create in Postman. The main difference is in how they are run and what the expectations around them are.

Contract tests are meant to establish a contract for how the API should work, so they need to be run primarily by the API provider. Consumers will sometimes run them to double-check the work that the provider is doing, but the main purpose of them is for the API provider to check that they are not violating the contract as they make changes to the API.

Since these tests are meant to verify things as code changes are made, these tests should be run as part of the build pipeline for the API development team. I covered how to run tests in a build pipeline in *Chapter 9, Running API Tests in CI with Newman*, so you can check that chapter out for more details on how to do this.

On the consumer side, you don't need to run the tests every time you make changes. Changes to the user interface do not affect the way that the API works. However, you might want to occasionally check that the API is still meeting the contract. One way that you can do that in Postman is with monitors. You can check out more details on how to use monitors by going back to *Chapter 10, Monitoring APIs with Postman*. Setting up a monitor to periodically check that the API is meeting the contract allows you to develop the application frontend with confidence, without needing to concern yourself with what the API development team is doing.

We all know that things need to change in software development. So, what do you do when things break? If you are doing consumer-driven contract testing, what do you do as the API provider when the contract tests fail?

## Fixing contract test failures

As with any test, contract tests will fail sometimes. They will fail for the same two main categories of reasons that other tests fail: a bug was introduced or the requirements changed. In the case of a bug, the code just needs to be corrected so that it does what the contract says that it should. In this case, fixing the test failure is straightforward: you just need to change the code so that it does the correct thing.

The other case is more difficult to navigate. Try as we might, the initial design of an API won't always be able to predict the needs of the customer in the long run. There may be times when an API needs to change in ways that break the contract. There may also be times when an API consumer needs the API to do something that is not in the contract yet. In both these cases, contract tests are valuable as they make it very clear that the API contract has changed.

They help ensure that the necessary conversations happen. However, this highlights the fact that the contract tests will probably have to change over time, and also that there are two (or more) different groups that might be interested in updating them. How do you go about making those changes to a shared source of truth like this? This is where collection-sharing becomes important.

## Sharing contract tests

For contract tests to work well, they need to be accessible by both the provider and the consumers. This applies both from the standpoint of running the tests and from the standpoint of maintaining them. I would suggest that contract tests are shared in their own workspace so that it is very clear that this is the location for storing those contracts. You can create and share a workspace by following these steps:

1. Click on the **Workspaces** dropdown at the top left of the application, as shown in the following screenshot:

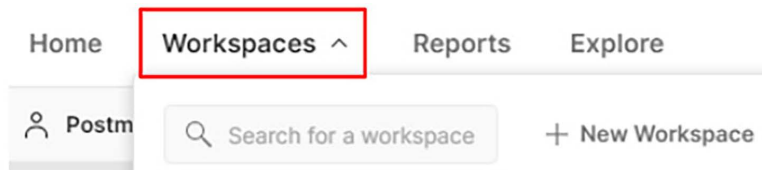


Figure 13.10: Creating a new workspace

2. Click on the **+ New Workspace** link.
3. Name the workspace something like **Contract Test Workspace**.
4. Since the purpose of this workspace is to have people working together as a team, ensure its visibility is set to **Team**.
5. If you wanted to, you could insert the email addresses of the people that you want to invite to join this workspace; then, you can click on **Create Workspace**.

Once you have this shared workspace, you can easily share your contract test collection to that workspace by following these steps:

1. Click on the **View more actions** menu beside the contract test collection that you want to share.
2. Choose the **Share collection** option.

3. Select the shared workspace that you created and click on the **Share and Continue** button.
4. If you want to assign different people on the team different roles for this collection (for example, to administer, view, or edit), you can do this on the next panel and then click on **Save Roles**.

This shared collection can now be viewed, edited, and run by anyone you give the proper permissions to. Setting things up in this way allows you to treat this test as a contract. The shared collection establishes a contract between all the users in this workspace. Anyone in that workspace can run the tests to verify that the API is doing what it should be doing. Also, if you want, you can allow people to make changes to the contract so that things can update and change as the requirements change over time.

**NOTE:**

The free Postman account has limits on how many requests you can share. If your team is doing a lot of contract tests and needs to share a lot of requests in this way, you might need to upgrade to a paid plan.

Contract testing is a powerful concept when put into practice correctly. This chapter should have given you a good foundation of how to do that, but don't forget that the most important thing is good communication. One of the main benefits of contract testing is that it makes it obvious when those conversations need to happen. When a contract test fails and needs to be updated, be sure to engage all stakeholders in a conversation about it. Software development is a very collaborative process, and the real power of contract tests comes in its ability to foster that collaboration.

## Summary

In this chapter, we did a deep dive into contract testing. You learned what a contract is and how you can use it. You then learned about the two different approaches to contract testing, which can either be consumer-driven or provider-driven. In addition to coming to a thorough understanding of the theory of contract testing, you learned how to use Postman to set up these kinds of tests. You also learned how to create a collection designed for contract testing.

I also showed you how to think about the tests that you add to a contract testing request and how to set up and use Postman Interceptor, which lets you figure out which requests and inputs make sense to include in a contract test suite.

I also showed you how to share contract tests in Postman so that they can be run and fixed in ways that embrace the usefulness of this testing technique.

With this under your belt, in the next chapter, we are going to put everything you have learned together into one big project that will help you put your Postman skills into practice in the real world!

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.com/invite/nEN6EBYPq9>



# 14

## API Security Testing

Security testing is its own area of specialization. It is probably worth an entire book all on its own. I'm not a security testing expert, but I think that every tester should at least have a basic understanding of this important topic. If possible, you should engage with security experts, since security breaches present one of the biggest risks to an API, but even if you do have access to them, there are some things you can do to at least establish a minimum bar for security in your application.

Perhaps you just want to do a sanity check before you have the security testing team look at your API. Perhaps you don't have access to security testing professionals. Whatever the case may be, in this chapter, I will help you get started with security testing. In doing so, I will discuss the following topics:

- The OWASP API Security list
- Fuzz testing with Postman

### **OWASP API Security list**

If you aren't a security testing expert, it can be hard to know what kinds of things to look for in security testing. One great place to start is with the OWASP API Security top 10 list (<https://owasp.org/www-project-api-security/>). I won't go through every item on the list, but let's go through a few of them to see how we might approach testing for them.

### **Authorization and authentication**

Many items on that list have to do with authorization and authentication. Of course, this makes sense, since authorization and authentication are at the core of any security strategy, but what kind of things can you check for to see if there might be weaknesses in your API's authentication and authorization approaches?

Let's start by looking at risk factors that could contribute to hackers getting through the authentication. Authentication is one of the first places for hackers to try, since users need to authenticate to get into a system, and if you can get the system to think you are a current user, you will have access to everything that users should have access to. Since authentication has to do with usernames and passwords, authentication hacks will often involve trying to guess these.

The simplest way to do this is via a brute-force attack, where the bad actor sets up a script that will try many combinations of usernames and passwords until it finds a combination that works. There are ways to mitigate this risk, and you should check that your API implements them. At the most basic level, a login flow for an API should have a rate limit on the number of login requests you can do in a period of time.

How can you test for this?

Well, a simple way to do it is to write a script that will repeatedly make login requests to see if you get hit by a rate limit. There are specialized tools that can help you with this, but you can also create a quick and dirty script to do this in Postman.

Let's look at an example of how to do this using the todo list app that we described in *Chapters 1* and *11*. If you haven't yet installed it, you can refer back to those chapters for how to do that. Once you have the app running, you can create a script to generate multiple login requests.

First of all, create a request that calls the todo list API. For this example, I will use the `/token` endpoint on the todo list application. You want to use a local service that you own so that you are not repeatedly calling someone else's API.

With the `/token` request setup, we want to try various values for the username and password. We can use some built-in fake data providers and set the password to `{{ $randomPassword }}` and the username to `{{ $randomUserName }}`.

POST

https://8000-djwester-todolisttestin-k75qq3a37q.ws-us106.gitpod.io/token

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

☐ none

☐ form-data

☒ x-www-form-urlencoded

☐ raw

☐ binary

☐ GraphQL

	Key	Value
<input checked="" type="checkbox"/>	username	{{ \$randomUserName }}
<input checked="" type="checkbox"/>	password	{{ \$randomPassword }}
	Key	Value

Figure 14.1: Random username and password

At this point, you could just repeatedly click on the **Send** button to try this out, but it would be nice to automate this. There are a couple of different ways that we could call this request multiple times.

One very simple way to do it would be to put the request into its own collection, and then run that collection. On the run configuration panel, set the number of iterations to how many times you want the request to execute for, run the collection, and then observe the results.

### Broken object-level authorization

One of the authorization-related security challenges for APIs has to do with correctly validating that a user is allowed to access the resource they are requesting access to. Sometimes, an API will validate the authentication (i.e. are you who you say you are?) but will not adequately validate the authorization (i.e. is the user allowed to access this resource?). If this incomplete validation is done at the object level, we can call that Broken Object-Level Authorization. This is the number one listed API vulnerability in the OWASP 2023 list, so it is a pretty important thing to take into account.



As an example, in the todo list application, there is an endpoint called `/user/admin` where a user can update their information. A user should only be able to update information about themselves and not be able to change other people's information. This endpoint implements authentication, and if you try to call it with an invalid token, it will reject your response. It can be tempting to think that if you tried that endpoint with an invalid token and it correctly rejected the response, you have tested it, but there would still be a big gap here. You also want to try testing it with valid credentials that belong to another user. This way, you can validate that it has the correct object-level authorization.

We need to make sure that the endpoint doesn't just validate that the credentials are valid credentials for some user in the system; it also needs to validate that the owner of those credentials is allowed to access the requested resource. As you can imagine, these kinds of issues can have very serious consequences. If someone can access or modify information they are not supposed to, they can potentially do a lot of damage to your system.

Testing this is also quite easy. For our example, the `/user/admin` endpoint, we could create a simple test like this:

1. Create a new collection.
2. In that collection, create a request to get a token at the `/token` endpoint.
3. Set the body type to `x-www-form-urlencoded`, and add a **username** with a value of `user1` and a **password** with a value of `12345` to the form.
4. Set the method to **POST**.
5. In the **Test** tab, add the following code to save the token to a variable:

```
var jsonData = pm.response.json();
token = jsonData.access_token

pm.collectionVariables.set("token", token)
```

6. Add a second request to the collection with the `/user/admin` endpoint.
7. On the **Authorization** tab, choose the **Bearer Token** type and set the token to `{{token}}`.
8. Send the request.

You should get back a `403 Forbidden` response. The `user1` user is not allowed to access the admin endpoint.

## Broken property-level authorization

A similar issue to not correctly protecting endpoints is allowing people to modify properties of an endpoint that they should not be allowed to change. This one comes in at third on the 2023 OWASP API security list.

An example of what is meant by this can be seen in the todo list API where one of the properties that a task has is the `created_by` property. This property tells you who created the task. You could imagine that we might want to only let certain users (maybe admin users) modify that. We don't want people to be able to change a task that they made to look like someone else made it, or to change someone else's task to look like they made it. Although all users should have access to the endpoint, in this scenario, we would only want a certain user to be able to modify that property on the endpoint.

In the example above, there isn't a lot of risk in allowing the wrong person access to the `created_by` property, but there can be more danger in other examples. How would you go about testing something like this? Well, much like the previous risk, you can quite easily check for this one by trying to modify a property as a user who should not have permission to do so.

We could use the same token request as above to generate a token for `user1` and perform the following steps:

1. Create a task by calling the `/tasks` endpoint with a POST action.
2. Set the **Authorization Type** to **Bearer Token** and the **value** to `{{token}}`.
3. Set the **Body** to **JSON** data and put in something like this for the body:

```
{
  "description": "Do Great Things",
  "status": "Draft",
  "created_by": "user2"
}
```

4. Send the request.

You can see that the request succeeds, and the task is created, stating that it was made by `user2` instead of `user1`. If it was important that people should not be able to modify the `created by` property, we would have a bug here.

## Unrestricted resource consumption

Although authorization and authentication are obviously very important parts of security, there are some other factors to consider as well. If someone is trying to attack your system, they might try to attack the underlying compute resources that run your API. Sometimes, systems have vulnerabilities that only appear when the system has used up all resources and crashed, while other times, the attacker may just be trying to harm the system or company by using up all system resources so that legitimate users can't use the system, or so that the company wastes a lot of money with unnecessary work.

These attacks are sometimes known as denial-of-service attacks and are a bit trickier to test for. Postman doesn't have much tooling to measure the load that an API call generates. Tools like JMeter are better suited for performance testing, but even in Postman, one thing that you can pay attention to is calls that are slow. A slow call is often slow because it does a lot of work, which is an indicator that it might consume a lot of resources. Postman shows you how long a request takes in the response area, so paying attention to that might give you an idea of which calls to focus on when trying to find calls that consume a lot of resources.

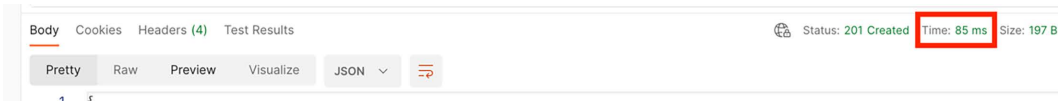


Figure 14.2: Call timing

Another way in which an API can be overwhelmed is by not restricting how many calls someone can make. Even if a call isn't that expensive, if someone tries to make millions of calls in a couple of minutes, it can still be overwhelmed. This is where API rate limits come into play. API rate limits determine how many calls any one user can make in a given period of time. Testing that they are there and working correctly is another way to mitigate the risk of unrestricted resource consumption.

Another thing to be careful of in this area is if you are calling any paid services. A lot of services now have usage-based billing, which means that you could end up with a very large bill if someone called one of your API endpoints that, in turn, makes calls to a third-party service. For example, imagine that every time you created a new user in your system, it calls an email service provider API, telling it to send that user a welcome email. The email service provider might bill you per email sent, so if someone got unrestricted access to the API endpoint that creates new users and created a million new users, you might get a very large bill from the email service provider the next month.

The biggest thing to test for when trying to mitigate the risk of excessive resource consumption is that you have limits in place. This could be limits on the rate at which any one client can make API calls, limits on how many times someone can call a given endpoint, limits on how many times a third-party service can be called before we raise an alert, or limits on how big the payload of a certain request can be. There are many different ways to set limits, but if you have an API that is used by external customers, you should think about which ones make sense to have and test that they are indeed set correctly.

## **Unrestricted access to business workflows**

The point of an external API is that you allow customers to access your systems through an API. However, you should think carefully about which endpoints you expose. Most modern applications will have some APIs that are internal. You might have some APIs that the frontend of your application uses to communicate with the server but you don't intend for customers to use. However, it is possible for someone using the UI of the application to use the developer tools (or other tooling) to watch the API requests that are sent. A malicious user could then use those endpoints to automate things that you might not want them to. For example, imagine a ticketing system that sells tickets for a high-demand event. If someone was able to figure out how the API works, they might be able to automate the purchasing of tickets and buy large quantities of them before other users, and then later resell those tickets at a large profit.

It can be difficult to test for these kinds of issues because, as a tester, you might have additional access for testing purposes that typical users don't have, but this is the kind of thing where black box testing can become important. Use the system as a typical user, and see if you can find any API endpoints that could negatively impact business workflows.

## **Unsafe consumption of APIs**

API testing isn't just about testing the APIs that your company makes but also the ones that it uses. Testing third-party APIs can be tricky, but one thing to think about is how much you should trust the data that you get from a third-party service. It can be tempting to assume that you can trust the data that you get from a trustworthy service, but even trustworthy services can get compromised or hacked. You should think about what kind of data protections you might want to have when consuming data from an external API.

For example, imagine you were working at an online education application that integrated with a third-party service, providing you with information about a student's grades at other institutions that they had attended. Imagine that the way it worked was that it would send you back a list of secure links that you could use to access the student's information at the various institutions.

You could just blindly trust the service and send requests to those links, but what would happen if that service was compromised and those links pointed to a hacker's website? You might be sending sensitive information or triggering other problems.

Always be careful when consuming information from a third-party API. You should have the same data scrubbing protections in place as you would for data you were processing that was input by users in the user interface.

Testing for this kind of thing can be a bit tricky. You would probably want to create a mock server that you could use to control the inputs. You could then have the requests to the third-party service go to your mock server, and you could try sending some bad inputs to see if your service correctly cleans up that data. You can read more about mock servers in *Chapter 12*.

There are other security threats listed in the OWASP top 10 list. I haven't gone through all of them, but feel free to check out the full list at <https://owasp.org/API-Security/editions/2023/en/0x00-toc/>. Hopefully, this brief overview that I've done has convinced you that you don't need to be a security testing expert to contribute to API security as a tester. There is certainly a place for dedicated security testers with specialized skills in this area, but every tester should be able to still do some basic security checks to help eliminate major flaws in an API.

I'd like to now turn to a couple of other testing strategies that lend themselves well to security testing.

## Fuzzing

Fuzzing involves providing invalid, unexpected, or random data as input to your program so that you can discover vulnerabilities and errors. The primary goal of fuzz testing is to identify security vulnerabilities, crashes, or other unexpected behavior in a software application. It is not a general-purpose testing technique and primarily helps you with identifying things that you might not have otherwise thought of or been able to test.

I suppose that technically fuzzing could be done manually by entering some pseudo-random inputs into a UI, but it is almost always done programmatically. There are some fuzzing tools like PeachFuzzer that can help you with this, but you can also create your own set of inputs if you want.

The way fuzzing works is that you start by generating a large volume of random or semi-random data to use as input. This input could include malformed or unexpected inputs that might not be handled correctly by the program. The point of fuzzing is that you don't have a lot of control over the input data – in other words, it is quite random.

You could put some boundaries on the data that you input. If, for example, you have already found issues with one kind of input, you might exclude that input from the input data that you create until the bug related to handling that data has been fixed. Fuzzing, as it was originally conceived, has **random input**. In this case, we would generate completely random data, although even here there are usually some boundaries on the types of data that are being generated – maybe bytes versus ASCII strings or Unicode strings. You might also weight your inputs to known “dangerous” characters like escape characters or quotation marks, etc. Generally, though, you want to discover things you wouldn’t have thought of on your own, so you want to keep the inputs random.

However, you can also have more limited forms of fuzzing like **mutation-based fuzzing**. This kind of fuzzing starts with existing known valid inputs, uses them to create variations, and injects them into the program. If your system already has good input validation, you might find that fully random fuzz testing doesn’t help you discover much because any invalid input is immediately rejected. In this case, mutation-based fuzzing might help you discover more information, as it lets you randomly explore the valid input space.

Once you’ve generated your data, you need to inject it into the system. You can do that through various interfaces, such as command-line arguments, file inputs, network protocols, or, most interestingly for us, APIs. APIs lend themselves well to this kind of testing, since they are easy to access programmatically, and it is easy to try many different inputs. Since fuzz testing is quite random, it can take a while to find issues, and you will often want to try many different inputs.

Of course, with fuzz testing, the inputs are random, so you can’t really have an “expected” value that the API will return. This means that you need to somehow monitor the API for any unexpected behavior, including crashes, exceptions, memory leaks, or security vulnerabilities. In this case, the aim isn’t to look for a particular response but, rather, to check that we don’t get a set of known bad responses.

Fuzz testing is particularly useful for finding security vulnerabilities, as it can reveal unforeseen attack vectors and weaknesses in the API’s processing and error-handling mechanisms. However, that’s all a little abstract, so let’s take a look at an example of creating a fuzz test in Postman.

## Fuzz testing with Postman

There are specialized fuzz testing tools out there, and some of them can even integrate with Postman. Covering how to use those is outside the scope of this book, but we can still do some fuzz testing directly in Postman. Let’s once again use the todo list application we set up in *Chapter 1*. We have used it a couple of times throughout this book, but if you have not yet set it up, you can check out *Chapter 1* to see the instructions on how to do so.

There are a lot of different inputs we could use for fuzz testing, but let's try fuzzing the description field when creating a new task:

1. In Postman, create a new request that points to the location of your todo list application and references the `/tasks` endpoint – something like this: `{{url}}/tasks`.
2. Set the request method to `POST`, and on the **Body** tab, choose the **raw** option and set the type to **JSON**.
3. Name the request `Make Task`.

The body of a request to create tasks looks like this:

```
{
  "description": "string",
  "status": "Draft"
}
```

For this example, let's try fuzzing the description field. We could try to create some kind of function that makes random noise for us to input into that field, but instead, let's use a pseudo-random approach. There is a GitHub repository called the *big list of naughty strings*. It contains a list of strings that often cause issues when they are used. Let's use that list as an input for the description field.

You can get a copy of it from the big list of naughty strings repo (<https://github.com/minimaxir/big-list-of-naughty-strings>). Either check out or download a copy of the repo.

The repo provides the naughty string list in a couple of formats, but none of them will directly work for use in Postman. However, we can easily convert one of the files to work for us. We will need to convert the `blns.base64.json` file, since the naughty strings in the non-encoded file will actually cause Postman itself to be unable to properly read the file. We can then convert this file to use key/value pairs instead of just being a simple list. The simplest way to do this is to open the file in a code or text editor and do a find and replace. Note that you will probably want to use your editor's *replace all* functionality instead of replacing one item at a time. Each line starts with two spaces, so you should search for two spaces and replace them with this string:

```
  {"naughtyString":
```

Each line also ends with a comma, so you can close the brace by doing a search for the commas and replacing them with this string:

```
  },
```

The last line in the file does not end with a comma, so you will also need to manually add a final close brace to the end of the file. Once you've done that, you can save the file and should now have a list of key/value pairs, where the key for each item in the list is `naughtyString`. If you are having trouble getting the file to format correctly, you can try using a JSON Formatter like the one you can find at <https://jsonformatter.curiousconcept.com/>, or you can download my version of the file from the GitHub repo for this course: <https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/main/Chapter14>.

Since we are using the base64-encoded version of the naughty strings file, we will need to decode those values before we use them. We can do this using the `atob` JavaScript library:

1. On the **Scripts** tab of the **Make Task** request, select **Pre-request** and import the `atob` library:

```
var atob = require('atob');
```

2. Now, get the naughty string variable from the input file and use the `atob` library to decode it:

```
var encoded_string = pm.iterationData.get("naughtyString")
var decoded_naughty_string = atob(encoded_string)
```

3. Set the decoded string as a variable that we can use in the request:

```
pm.variables.set("naughtyString", decoded_naughty_string)
```

We can now use that variable in the body of our request:

1. In the body of the **Make Task** request, change the description to use the `naughtyString` variable:

```
"description": "{{naughtyString}}"
```

It is great that we can fuzz the API with inputs like this, but how would we know if something went wrong? One way is that the API might just crash from one of the inputs and the run would exit before it is completed, but there might also be other issues that this might reveal as well.

With fuzz testing, we don't control the inputs too much, so we don't necessarily know what to look for in the outputs. Instead of checking for specific return values, with this kind of testing we need to look at more generic ways that things might have gone wrong.



In this case, we could assume that every call should successfully create a task and add a test to check that the status code is 201.

2. On the **Scripts** tab, go to the **Post-response** and put in this test:

```
pm.test("Status code is 201", function () {
  pm.response.to.have.status(201);
})
```

3. Save the request, and then go to the collection that the request is in.
4. Click on the **Run Collection** button.
5. Make sure to deselect any other requests that might be in that collection, leaving only the **Make Task** request.
6. From the right-hand panel, click on the **Select File** button and browse to the `blns.base64.json` file that you just created.
7. Click on the **Run collection** button, and it should attempt to create a task for each line in the big list of naughty strings.

You will notice that some of the tests fail with a 422 Unprocessable Entity error. This seems like a reasonable error to fetch in the case of bad data like this. We can also look through the failed request and see if there are any interesting things to note or any themes. You can do this by clicking on the **Failed** tab of the run results, clicking on one of the failed runs, and then selecting the **Request** button.

todo-book-testing - Run results

Run today at 07:35:24 · [View all runs](#)

Run Again Automate Run + New

Source	Environment	Iterations	Duration	All tests	Avg. Resp. Time
Runner	Contract Test Environment	676	1m 14s	676	87 ms

All Tests Passed (396) **Failed (280)** Skipped (0)

Iteration 17

**POST Make Task**

<https://8000-djwester-todolisttestin-s4xo8esumtn.ws-us107.gitpod.io/tasks>

422 Unprocessable Entity

FAIL Status code is 201 | AssertionError: expected response to have status code 201 but got 422

Iteration 91

**POST Make Task**

<https://8000-djwester-todolisttestin-s4xo8esumtn.ws-us107.gitpod.io/tasks>

422 Unprocessable Entity

FAIL Status code is 201 | AssertionError: expected response to have status code 201 but got 422

17 POST todo-book-testing / Make Task

18 Response Headers **Request**

19

20 Pretty

21

```

22 {
23   "description": "<script>{}|_+",
24   "status": "Draft",
25   "created_by": "user2"
26 }
27
```

Figure 14.3: Viewing failed request results

I noticed a couple of interesting things when looking through these results. One was a description that a simple slash (/) failed.

That seems like an interesting special character to fail, and if I tried to do some deep testing of this application, I might want to dig into that a bit more to try and understand what is going on with it.

Another interesting thing I noticed was that many of the failed requests seem to try to use HTML. It seems like a lot of HTML is un-processable. This is probably a good thing, as HTML can be a dangerous thing to allow in inputs since unsafe parsing of HTML inputs can lead to dangerous injection errors. However, if you go to the UI after running this fuzz test, you will see that we get an alert that says 123. This is because one of the inputs gets past whatever is causing the site to reject some of the other HTML inputs and creates an injection error. You can imagine that allowing users to create arbitrary popups could be confusing or dangerous, and also that being able to do this means you could probably do other more nefarious things as well. If we found something like this in a real application, we would want to raise a high-priority security bug, as this can be quite dangerous.

## Cleaning up the tests

You can see that this kind of testing can give you a lot of insight into things that you might not otherwise have thought of, but depending on how your testing environment is set up, one thing that can be annoying about fuzz testing like this is all the data that gets left behind. However, since we are automating an API, we can automate the cleanup as well. Let's automate deleting all the tasks in a system:

1. Create a new collection called **Cleanup Collection**.
2. Add a variable to the collection, called `url`, and set its value to the value of the todo list site **URL**.
3. Go to the **Authorization** tab for the collection and set the type to **Bearer Token**, and in the **Token** field, enter `{{token}}`.
4. Save the collection.
5. Create a new request in the collection called **Token** and set the type to **POST** and the **URL** at `{{url}}/token`.
6. On the body tab, set choose the **x-www-form-urlencoded** option, and then create a key called `username` with a value of `user2`, and a key called `password` with a value of `12345`.

Note that the username to get the token needs to be the same as the one used to create the tasks. I have set it to use `user2`, assuming that this is the user that created the tasks. If you used a different user to create the tasks, you will need to specify that username instead, since the system will not let you delete other users' tasks.

You can see which user created a task by looking at the **created\_by** field when getting the list of tasks.

7. Go to the **Scripts** tab, and in the **Post-response**, save the token as a collection variable with the following code:

```
var jsonData = pm.response.json();
token = jsonData.access_token

pm.collectionVariables.set("token", token)
```

8. Save the Token request.
9. Add another request to the collection – this time, one that will list all the tasks. Make sure it is a GET request and give it the **URL** `{{url}}/tasks`.

We are going to need to get the list of task IDs from here so that we can iterate over it and delete each one.

10. In the post-response to this request, add the following code:

```
var jsonData = pm.response.json();

var task_ids = []
for (let x in jsonData){
  id = jsonData[x].id;
  task_ids.push(id);
}

pm.variables.set("task_ids", task_ids);
```

This code gets all the task IDs from the response, puts them into an array, and then makes that array available as a variable.

11. Add a **DELETE** request to the collection and specify the **URL** as `{{url}}/tasks/{{id}}`.
12. In the **Pre-request Script** tab for this request, get the list of task IDs using the following code:

```
var task_ids = pm.variables.get("task_ids");
var id = task_ids.pop()
pm.variables.set("id", id);
```

```
var last_iteration = false;
if (task_ids.length == 0) {
    last_iteration = true;
}
pm.variables.set("last_iteration", last_iteration);
```

This code gets the list of task IDs that were set in the previous request, and then it removes the last ID from that list, storing it in the `id` variable so that it can be used in the request URL as the variable that we are going to delete. It then checks if this is the last variable in the list, and if it is, it sets a variable called `last_iteration` to `true` and makes this variable available for use.

13. Go to the **Tests** tab for this request and set the following code:

```
var last_iteration = pm.variables.get("last_iteration")
if (last_iteration == true){
    postman.setNextRequest(null)
}
else{
    postman.setNextRequest("Delete")
}
```

This code gets the `last_iteration` variable that we set in the pre-request tab, and then uses its value to determine what we should set the next request to be. If we are on the last iteration, we set it to `null`, which means that we are done and Postman will leave this request, but if we are not on the last iteration, we set it to the name of the current request. This will cause Postman to rerun this request. It will go back to the pre-request script, get the next ID, delete that one, and then check again if it is done yet. This will continue until all the tasks have been deleted.

14. Make sure that all the requests are saved, and then run the collection. You only need to specify one iteration, since the `Delete` request will loop as many times as needed inside of the run.

It will take a little while to run, due to the number of tasks that need to be deleted, but you should see that it eventually cleans up all the tasks you created with your fuzzing test.

## Fuzzing with built-in methods in Postman

We've seen how to do fuzz testing in Postman with an external data source, but we could also leverage some built-in functionality to fuzz test. Postman can create a number of different kinds of random data, and we can use those random data generators to set up some fuzz tests.

When we previously created the request to delete the tasks, we made it so that this request could run in a loop. We can use similar logic to create a test that will call the API with random data multiple times:

1. Duplicate the collection you used to create the big list of naughty strings tests.
2. Remove all the code from the **Pre-request Script** tab.
3. On the **Body** tab of the **Make Task** request, replace the `{{naughtyString}}` with `{{randomLoremSentence}}`.
4. Save the request, and on the collection page, click on the **Run** button.
5. On the run collection page, set the number of iterations to, say, 3, and then run it.

The collection will run the request three times, and each time, it should have a different description. However, this isn't a great fuzz test because, although it is random, we know that the type of input it gives is the type this field expects to get. If we wanted to make this a little more random, we could try using some other random variables, like `{{randomNatureImage}}` for example. The point of fuzz testing is to help you find things that you might not know about otherwise, so you want to make sure you try with inputs that should be invalid.

We can extend this to take a random sample of the random variables by using the JavaScript `lodash` library. This library is included with the version of JavaScript used in Postman and can be accessed with an underscore:

1. On the **Pre-request** part of the **Script** tab, save a variable that will select from a list of random variable generators by putting in this code:

```
pm.variables.set('description', `${_.  
sample(["{{$randomAbbreviation}}", "{{$randomAdjective}}"])}`);
```

This code will randomly select one of the random variable generators and set the value it gives as the description variable.

2. You can use the description variable on the **Body** tab by setting the body like this:

```
{  
  "description": "{{description}}",  
  "status": "Draft",  
  "created_by": "user2"  
}
```

If you want to extend the number of random variable generators that you pick from, you merely need to add more items to the list being passed to the sample method. You can find the full list of available random variable generators here: <https://learning.postman.com/docs/writing-scripts/script-references/variables-list/>.

## Summary

In this chapter, we learned a few approaches and techniques that you can use to do security testing in Postman. This kind of testing is not a replacement for the work of professional security testers, but it should help you find some basic issues. We learned about some of the most common security risks and mistakes in APIs, and we discussed how to set up tests in Postman that will help you find these issues. We used the OWASP top 10 list as a guide to what kind of issues to look for.

We discussed fuzz testing as a technique that you can use to introduce randomness to your testing. This randomness can help you find issues that you might otherwise never have thought about. We learned about the big list of naughty strings and how to set up a Postman collection that can use that list to check if your API has poor behavior, in response to any of those nasty strings. Since the point of fuzz testing is to try many random things, we also went over how to clean up a system after we are done. We also saw how we can leverage some of the built-in functionality that Postman has with its random input variables to create fuzzing inputs.

Security testing is a topic that can (and does) have entire books written about it, so we have only been able to present a few ideas here to help you get started, but what we have presented should help you find some security issues in APIs that you test. If you are on a team that doesn't have access to security testing professionals, or if you just want to do some more due diligence before having the security testers look at your API, you can use some of the techniques in this chapter to help you do just that!

Another area of specialty in testing that is worth having a baseline knowledge of is performance testing. In the next chapter, we will look at performance testing APIs.

## Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://discord.com/invite/nEN6EBYPq9>



# 15

## Performance Testing an API

APIs are meant to be interacted with programmatically. This means that access to them can be automated. This ability to automate is great and is one of the reasons that APIs have taken off the way they have. However, the ability to automate them means that they can be, deliberately or inadvertently, interacted with in ways that can cause performance issues.

Performance testing is an area of testing that has its own specialists. There are a huge number of considerations and a lot of complexity that go into performance testing. I have done some performance testing in my career and worked with performance testing teams, but I am not an expert on this topic. However, as with security testing, I believe that this topic is too important to API testing for us to ignore. We might not be able to deeply test for performance, but as well-rounded testers, we want to be able to check for some of the more obvious things.

In this chapter, we will learn about some of the basic performance testing ideas and techniques, and then we will look at what kinds of things we can test in Postman. There are many other tools that are specifically designed for performance testing, but since we are just getting our toes wet here and trying to make sure we understand the basics, we will stick with Postman.

In this chapter, we will cover the following topics:

- Different types of performance load
- Using load profiles in Postman
- Running performance tests in Postman
- Performance testing considerations



## Different types of performance load

Performance testing is a deceptively large term. There are actually many ways to think about performance testing. At the heart of the term is, of course, the idea that we are checking to make sure the system is performant. Ultimately, what we care about is that when a user requests a certain action, we are able to give them the response they want in a timeframe that is reasonable to them. However, where performance testing gets complicated is thinking about the different things that can cause a system to slow down. Let's look at some of the different types of load that a software system might have.

### Processing load

Processing load refers to the load on the processing units of the hardware that your service runs on. This is when you are trying to do something that is computationally difficult. There are many kinds of actions that can be expensive to compute. It might be trying to render a large image or video, trying to search through a large database, or even just a bug that triggers some kind of inefficient looping. Whatever the trigger, this type of load slows down the CPU or GPU of a system.

In the context of an API, this type of load can be generated in a couple of different ways. There might be certain endpoints in your API that request computationally expensive things. For example, you might have an endpoint that generates a report, which takes a bit of processing to build in the backend. When testing for this kind of load, you want to make sure that the system can handle these requests, but you might also want to be able to find out which requests are expensive. When you explore an API in Postman, it will report the time that it took to get a response from the request at the top of the response section in the UI.

---

 Status: 200 OK Time: 651 ms Size: 762 B

*Figure 15.1: Request time*

Long request times don't necessarily mean that the request causes a lot of computation; it could be that there was some other request that used up the CPU and blocked or slowed down this request, or it could be that there was a networking issue. However, a slow response time is a good indicator that it is worth investigating the performance of this request.

Sometimes, a request, like a search request that queries a database in the backend, can perform fine initially but gets slower as the database complexity grows or as changes are made in the backend. In those cases, it would be nice to know that things are getting slower.

Sometimes, changes in one part of a system can inadvertently cause issues in another part of the system, and you might not know about it until the changes are in production and cause a heavy load on your production services. You can check that a response time isn't becoming too slow in Postman.

If you go to the **Post-response** section of the **Scripts** tab of a request, you can add a test like this to any request:

```
pm.test("Response time is less than 5 seconds" function(){  
  pm.expect(pm.response.responseTime).to.be.below(5000);  
});
```

This will fail the test if it takes longer than 5 seconds to complete the request. In this example, I used quite a long request time. Depending on what your service does and how long you expect the request to take, you will want to adjust this time. There can be a lot of things that can cause requests to occasionally be slow. You don't want to get a test failure every time there is a network slowdown, or some increased load on the testing server from other tests. Be careful to set your expected time high enough that it won't get triggered too often by things you don't care about, but also try to keep it low enough that it will let you know when there are real issues before they become so serious that they cause production outages.



#### Setting an Expected Response Time

A good rule of thumb to set an expected response time is to run a request a few times to get a feeling for how long it usually takes, and then approximately double that time as the upper limit on how long it should take to execute the request.

It is also possible to add a test like the one above to the **Post-response** section of the **Scripts** tab of a collection so that it will automatically check the performance of every test in the collection. This makes it a bit tricky to set the time level appropriately, but it has the benefit of adding that check to every test in the collection without you needing to think about it.

## Memory load

Another type of load that a system can experience is memory load. This can be in the form of RAM, where a request causes a lot of information to be loaded into the temporary memory of the service, or it can take the form of filling up the hard drive memory.

Overloading the temporary memory of a system usually happens when something that is computationally expensive tries to manipulate something in memory. Unless your system has only a small amount of RAM or you have a bug in your service, it isn't very likely that one request will overwhelm the RAM on your system. This can make it a bit harder to detect when a request might be causing issues. Unless you can see the hardware monitoring of the system, you might not know how much temporary memory a request uses. Postman itself doesn't have features that let you directly look into what is going on in the memory of your service, but if you suspect there are issues related to this, you should work with your operations team to get some insight into the memory utilization.

If the memory load involves filling up the hard drive, you might be able to make an educated guess as to where the problems might be. For example, if you have an endpoint that allows users to upload a file, you could check if the server appropriately limits the size of the file that you can upload.

## Connection load

Connection load is an increase in the number of connections to your service. If you have been around on the internet long enough, you might remember the so-called Slashdot Effect. This described how websites would tend to go down or have issues, due to sudden surges in traffic if an article linking to the site was posted on the popular Slashdot news service. The surge in traffic would overload the server, and it would become unresponsive or even crash.

This kind of load is what I refer to when I talk about connection load. It isn't fully distinct from processing and memory load, since it sometimes happens that connection load causes a memory or processing overload. However, connection load can also be a problem in its own right. Even if the server can handle processing a large number of requests at a time, there is still a limit to how many connections a server can have open at one time.

One nice thing about testing this kind of load is that you can have a lot of control over the number of connections that you send to a server. There are some specialty performance testing tools like JMeter that are specifically designed to generate this kind of load. These tools enable a lot of interesting connection load tests and are worth investigating if you want to go more in-depth with your performance testing.

## Using load profiles in Postman

Although Postman doesn't have as many performance testing features as tools like JMeter, it does support some basic performance testing for connection loads. It provides several different types of load profiles that can be used to simulate different ways that you might get connection load on your site.

### Fixed load profile

The simplest form of connection loading is a fixed load profile. With this profile, we simulate a steady stream of users connecting to your site. You might use this kind of profile in a baseline test to establish how well the site functions under business-as-usual conditions.

You can create this kind of profile in Postman with the following steps:

1. Create a collection called **Performance Testing**.
2. Create a new request in the collection and set the **URL** to `{{url}}/tasks`.
3. On the collection, create a variable called `url` and set its value to point to the location of the todo app mentioned in previous chapters. You can go back to *Chapter 1* and follow the instructions there to set it up if you haven't already.
4. Save the collection and the request.
5. Go to the collection and click on the **Run** button.
6. In the right-hand panel, switch to the **Performance** tab.

You can now see some of the performance settings that Postman has:

Functional Performance

Test how your APIs perform under load

Simulate real-world traffic from your local machine and observe the performance of your APIs. Learn more about [performance testing](#)

Set up your performance test

Load profile ⓘ  
Fixed

Virtual users ⓘ  
20

Test duration  
10 mins

20 VUs

010 mins

Simulate 20 virtual users repeatedly running the collection, in parallel, for 10 minutes.

Data file ⓘ FEATURE TRIAL  

Select file

Run

Figure 15.2: Performance run

You can select a load profile and then select how many users you want to simulate, along with how long you want to run the test for.

Let's set up a small test with a **Fixed** load profile, **10 Virtual Users**, and a **Test Duration of 1 min**. Once you have input those settings, choose to run the collection. Postman should run the test and give you a report that looks something like this:



Figure 15.3: Performance testing report

You can see in the example that I ran that the initial requests were a bit slow as the server warmed up, and then after that, they were pretty stable. This is a good pattern to see. The responses don't slow down over time, and although there is some slight variation along the way, the response times are pretty consistent as well. However, you can also see that there were a few errors during the server warm-up. This would be a more concerning pattern if we had an application that was expected to scale across servers. We wouldn't want there to be errors every time we spun up a new instance of our application.

You can see in the **Load profile** dropdown for the performance test runner that there are several different ways to generate load. Let's take a more in-depth look at some of these load profiles.

## Spike load profile

A spike load is meant to simulate a sudden increase in the traffic to a website. In a spike load, the traffic starts at a somewhat low baseline, very quickly rises to a much higher level, and then, shortly after, drops back down to somewhere close to the baseline.

If you choose the **Spike** load profile in Postman, you can see an example of what a spike load might look like.

Set up your performance test

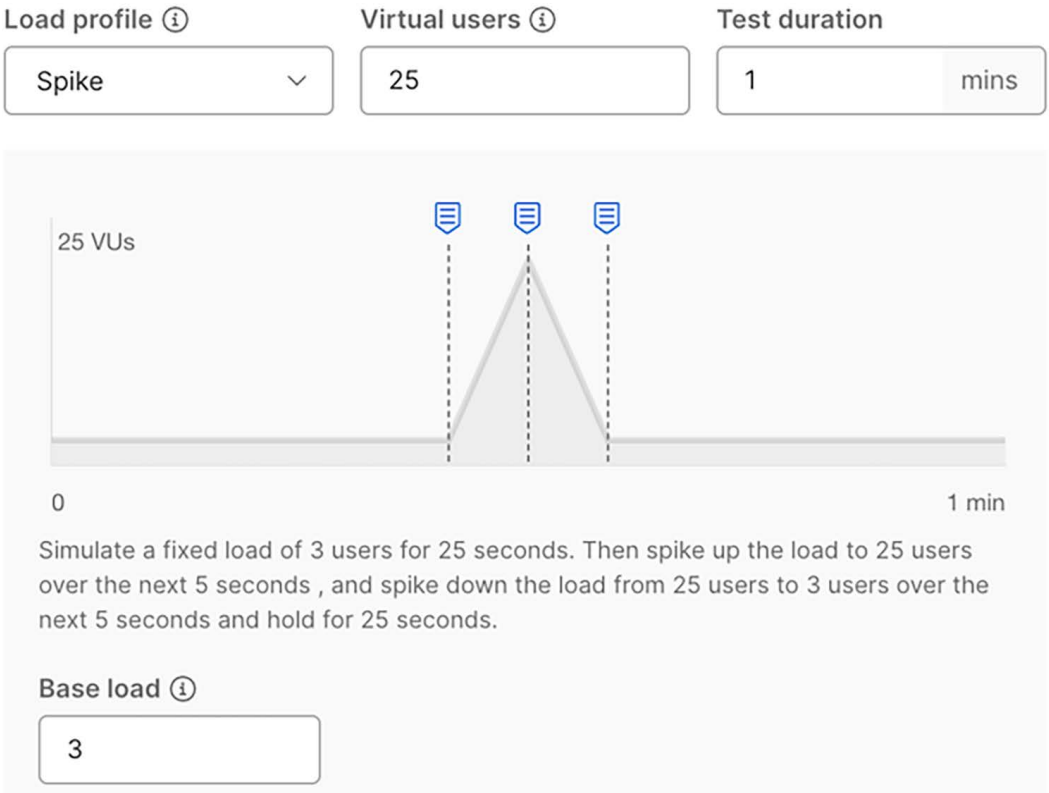


Figure 15.4: Spike load profile

Spike loads can be difficult for a server to handle, even if it has some autoscaling policies in place. If the increased load happens too quickly, the server might not be able to spin up new instances quickly enough.

You can play around with how big the spike is and how quickly it occurs by changing the settings in Postman. If you run the collection, you can see that it will first run with the number of users specified in the **Base load** for a while, then very quickly add more virtual users, and then scale back down to the base load.

As with all performance testing, when spike testing, you need to be aware of how your testing servers might differ from production. For example, are there different scaling policies on the test servers? In other words, will they add and remove servers in the same ways? Another thing to be aware of is the size of the servers. Often, test instances will run on less powerful hardware than production. This can lead to them responding differently.

You need to understand what issues you are looking for. If you are trying to see how quickly a service can respond in terms of starting new virtual machines when there is a sudden spike in the load, you will need to make sure that the settings around auto-scaling match between the test and production instances of your service. However, it may be OK to have a smaller base virtual machine so that you hit the scaling point sooner. It is important that you know something about the operational settings of test systems and how they match up with the production system; otherwise, you might find issues that only show up in the test system and don't affect production.

## Ramp load profile

Another type of load profile is a ramp load. With this type of load, you once again start with a baseline number of users, but then you gradually ramp up the number of users until you reach a new, higher, baseline. Compared to a spike load, this load increase is more gradual and then it stays at the higher level for some time.

This kind of load is common for services that have many of their users in a certain geographic region. When it is nighttime in that region, there are less people using that service, and so the baseline load drops, but then, in the morning, the number of users gradually ramp up to a much higher daytime usage level, staying consistent at that level throughout the day.



A ramp load profile might look like this:

### Set up your performance test

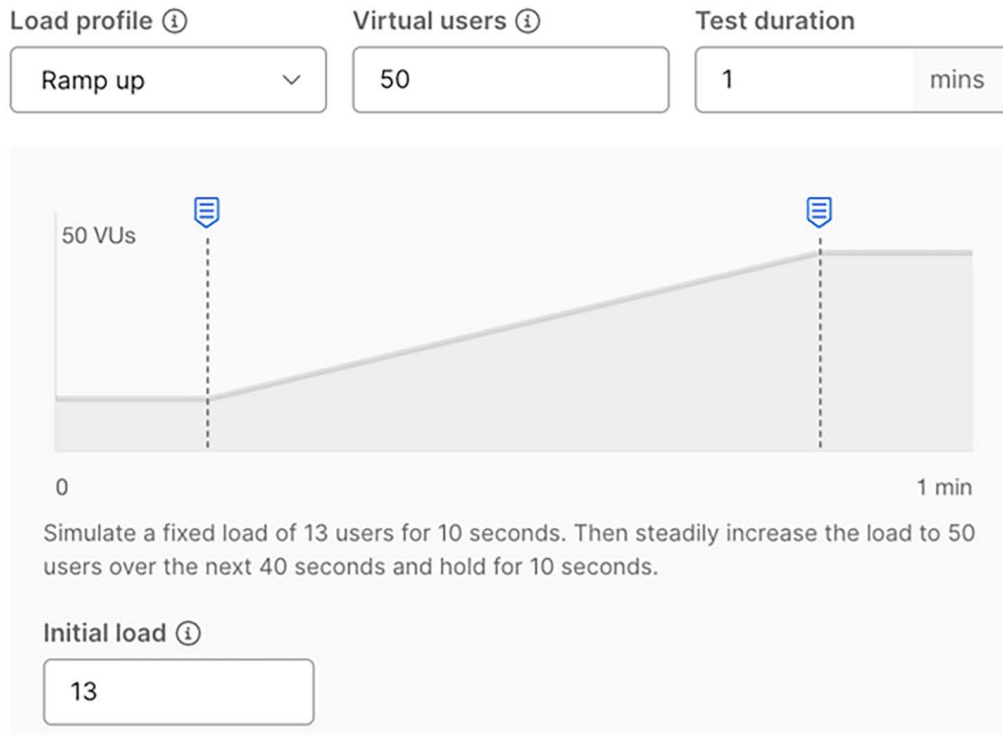


Figure 15.5: Ramp load profile

With this kind of load profile, you will ramp up the number of users more gradually than you would with a spike profile. With a spike profile, you usually try to see how quickly a service can add and remove resources in response to sudden changes in the number of users. With a ramp profile, you usually try to see how well the service can move to an entirely new baseline. Sometimes, a service will want to reduce the resources back down to the initial baseline level, so it can struggle to maintain things at the new, higher, baseline.

You can also use this kind of profile to **test for failure**. Testing for failure involves increasing the number of requests until something fails in the service. With this type of testing, you try to see which part of a system is the first to break. Does it run out of memory, does the processing power die, or does the database get so slow that it locks up? Or is it something else entirely?

If you wanted to do that kind of test, you would probably make your ramp profile look something like this:

### Set up your performance test

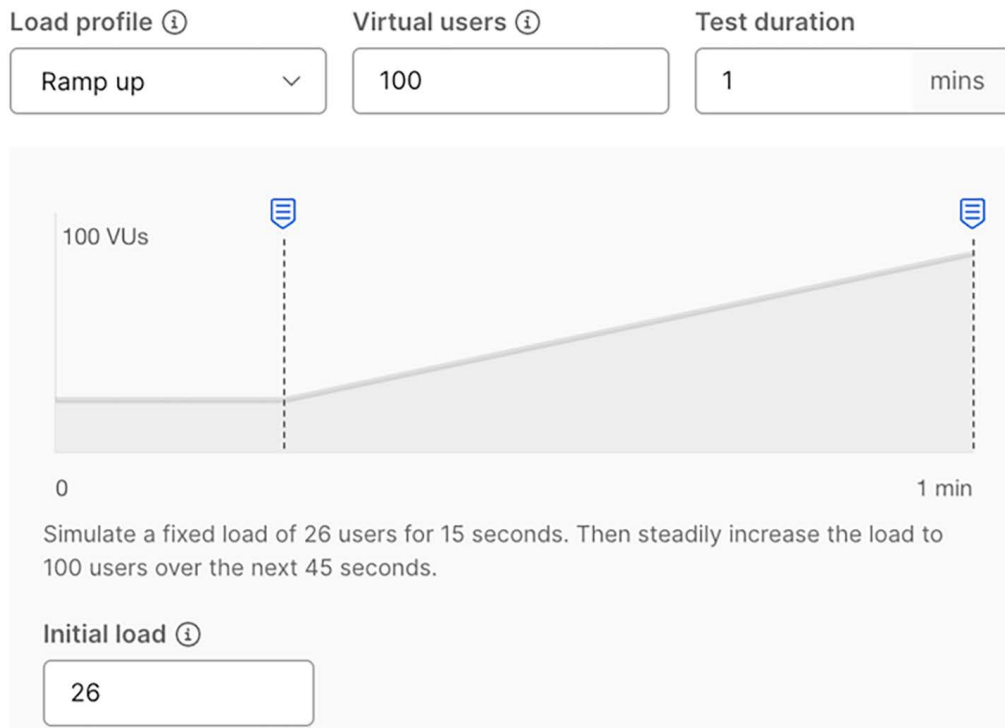


Figure 15.6: Ramp to failure

The Postman performance feature only lets you go up to 100 virtual users, so if you wanted to really run a test like that where you just keep increasing the number of users until something fails, you would probably need to use a tool that allows you to go past 100 virtual users. It is unlikely that a service would start to fail with only 100 users.

## Endurance load profile

An endurance load profile is a kind of ramp profile that stays at very high user level for a long time. The purpose of this kind of testing is to see if there are any long running processes that might gradually increase their resource consumption over time. An example of this might be a memory leak bug where a process very slowly increases its memory usage over time.

You might not notice this memory usage in a normal testing scenario, but if you run at a high load for hours, you can notice processes that consume a large amount of memory. Postman doesn't have an endurance load profile, but you can set it up using the Ramp up profile. It would look something like this:

### Set up your performance test

Load profile ⓘ

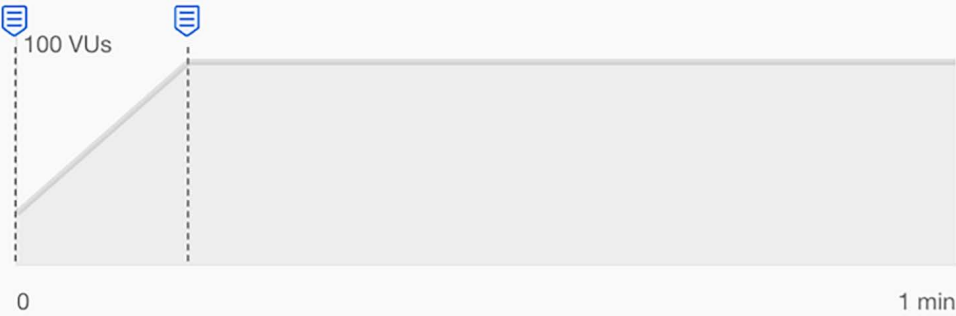
Ramp up ▼

Virtual users ⓘ

100

Test duration

1 mins



Steadily increase the load to 100 users over 11 seconds and hold for 49 seconds.

Initial load ⓘ

25

Figure 15.7: Endurance load profile

Once again, the maximum load of 100 virtual users is probably not enough for a good endurance test, but it is still good to be aware of this kind of test. Even if you were to run only 100 users for an hour, you might be able to find out some interesting things about how your system responds.

## Running performance tests in postman

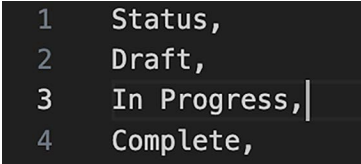
Now that we have covered some of the different types of performance testing, let's try running some performance tests in Postman. You can run a very simple performance test by going to a collection, clicking on **Run**, and then, on the **Performance** tab, setting up a simple load profile like one of those discussed above and hitting **Run**.

When running this kind of load, Postman will use the same input data for each of the different users that it simulates. However, if you want a bit more control and variation in the data, you can create a data file to control this. We will use the todo list application mentioned earlier:

1. Create a todo list item in the application and get the ID of it (for this example, I will use an ID of 1).
2. Create a collection, and in the collection, create a request.
3. Set the **URL** to `{{url}}/tasks/1` and the type to **PUT**.
4. Add this code to the **Body** of the request:

```
{
  "description": "Do Something Great",
  "status": "{{Status}}"
}
```

5. Create a `.csv` file that looks like this:



```
1 Status,
2 Draft,
3 In Progress,
4 Complete,
```

Figure 15.8: Performance test inputs

The first row in the file should match the variable name that you specified in the body of your request.

1. Click **Run** on the collection and go to the **Performance** tab on the run panel.
2. Set up a performance test by choosing the **Load profile**, the number of virtual users, and the test duration that you want.
3. Under **Data file**, choose the **Select file** option and browse to where you saved the `.csv` file you created.
4. Change the **VU data mapping** setting to **Randomize**. This tells each virtual user to pick data from a random row when it runs the collection.
5. Click on **Continue** and then **Run**.

At the time of writing, the **Data file** option is available in the free version of Postman, but it might eventually become a paid feature. However, there are some other ways you could use different data for your different virtual users.

For example, you could use some of the built-in random methods that Postman has. You could modify the PUT request created above to include this in the body:

```
{
  "description": "{{${randomWords}}}",
  "status": "Complete"
}
```

Now, if you run a performance test, each request sent will have a different description.

## Running multiple requests

So far, we have only looked at running performance tests against one request at a time, but we can also run against a collection with multiple requests:

1. In the performance collection you previously made, copy the PUT request.
2. Rename the copied request **Create a Task** and change the method to POST.
3. Drag the request up so that it is the first request in the collection.
4. On the **Scripts** tab, put the following code into the **Post-request** section to save the task ID to a variable:

```
var jsonData = pm.response.json()
var task_id = jsonData.id
pm.collectionVariables.set("task_id", task_id)
```

5. Make sure to save the request, and then go to the collection and run it.
6. Set the **Performance** setting to some smaller value (say, 10 virtual users and a 1-minute duration) and run the test.

Once the test has finished running, the chart shows you the average response time of the requests. This is aggregate data. You can also see the data on a per-request basis by using the **Filter by requests** dropdown at the top of the chart.



Figure 15.9: Filter by requests

Let's add one more request to the performance test collection – this time, a request to GET the full task list:

1. Create a new request in the performance test collection and call it `Get all Tasks`.
2. Leave the method as GET and set the URL to `{{url}}/tasks`.
3. Make sure it is the last request in the collection and run the collection.
4. This time, on the performance test setting, choose a slightly higher value for the number of virtual users (say, 25 or so). You can keep the run duration short.
5. Run the test.

As the test runs, it will create more and more tasks, so we would expect that the request to get all tasks would be a bit slower as the number of tasks returned slows. If you filter down to that request, you should indeed see that happening.

Now, try running the collection again but, this time, with the maximum number of users, `100`. You will probably notice that the run times for all the requests are significantly increased from when we run with only 25 users. It seems that the server is starting to get bogged down at this level. When I ran this test, I also started to see some errors. When you run into errors like this, you generally will want to debug them. To do this, you can click on **Errors** at the top left of the panel:

Summary Errors

Figure 15.10: Errors tab

This shows you a chart that plots the number of errored requests of each type that occurred as the run progressed. In my case, all the errors were 500 errors, so the chart looked like this:

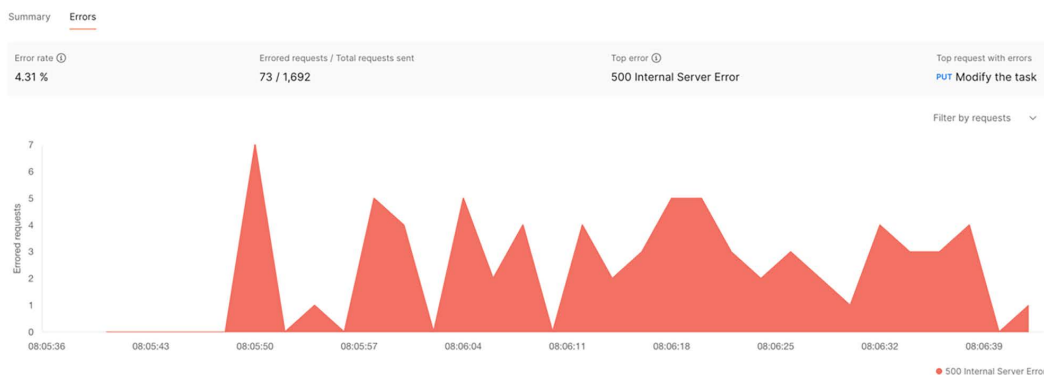


Figure 15.11: Error chart

You can also expand the errors in the error class section and then click on one of the requests to drill into more detail on what happened with the failure. You can see what the error response was, and you can also check on the headers and body of the request that was sent.

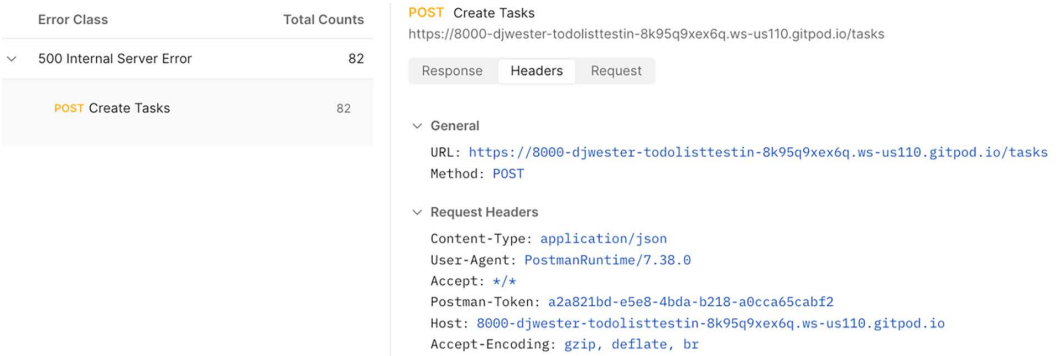


Figure 15.12: Debug information for errored requests

Information about the details of the request and response can help you ensure that the requests were set up correctly. In this case, it looks like they were, and the server crashed due to too high of a load. We can see from the chart that not all of the requests crashed. The highest peak seems to have been at about 7 errors, and since we were running 100 virtual users, this is only a small percentage of the requests.

At first glance, this doesn't seem to give us too much information, but there are a couple of things that we can learn from this. The first thing that we can learn is that although the server returns some kind of internal error, it seems to be able to recover from those errors. Whatever the error is, it doesn't take the whole server down, as other requests seem to be able to successfully process after the first errors occur.

Another interesting thing here is that all the errors apply to the POST and PUT requests, but the GET request did not hit a single error. This information hints that the problem might not be with the hardware getting overloaded or the server not being able to process that many requests at once. If those were the cause, we would expect to see the errors distributed across all the requests. The fact that it is only requests that create or modify things that throw errors hints that the problem might be in the database processing. Until a database gets large, retrieving data from it is significantly less resource-intensive than modifying it. If I was to make a guess about where to start looking for this issue in the code, I would guess that we should start by looking at how well the database layer can handle multiple concurrent requests to add or modify data.

This example shows you how even simple performance tests can reveal interesting things about your application. You don't have to be a performance testing expert to find out some useful information with simple performance tests.

## Performance testing considerations

When should you do performance testing? What kind of data should you use in your tests? How will you know if they are too slow? So far in this chapter, we have focused on how to set up and run a performance test, but now, I want to look at a few considerations to make a *good* performance test. What are some of the things that you should keep in mind when you create a performance test so that you can maximize the amount of information that you can get from the test?

### When to do performance testing

The question of when to run performance tests is a tricky one. Should you run them early in the development cycle so that you can adjust your code to be more performant early on, or should you wait until later in the cycle so that you don't waste time running tests on code that is not yet complete, as it is going to change anyway?

In other words, this is a software development trade-off between not wanting to write code that you don't currently need and knowing that it can be a lot of work to rewrite code that wasn't designed for the current situation. As with most things in life, I think there is a balance to be struck here.

Early on in the development process, you probably want feedback on whether the approach you take is going to be scalable to at least one level beyond what you expect. You don't want to spend a lot of time trying to design and test for a system that can handle millions of requests per minute if you only expect to have a couple hundred users initially. However, I would argue that even if you only expect a couple of hundred users, you should test at a level beyond that (maybe a few thousand users). The goal is to make sure that you are ready for the next level of growth should it come. If you don't prepare for some growth, it could cause you real pain if it did happen. Improving performance can be quite tricky, and that problem is made much harder if you need to do so under the intense time pressure of a system in production that fails to keep up with the current load. You don't want to spend too much time over-optimizing your system for loads that might never occur, but you also want to work slightly ahead of anticipated loads so that you can respond to unexpected increases.



Keeping all of that in mind, when should we run performance tests? Probably quite early in the development cycle. If we know that we are anticipating a certain load, we can create tests that simulate a load that is a bit more than that and start running them early on in the process. If the code architecture is such that we can't handle that load, the earlier we know about it, the easier it is to change it.

I would even argue that it is a great idea to include performance testing in your build or **continuous integration (CI)** runs. It's no fun to have someone make a change that is logically correct but accidentally introduces a major slowdown to a system. One thing to note here is that this would probably require a different kind of performance test than the ones we've talked about so far. If you run a test multiple times a day, you probably don't want to do the kinds of load testing that we've talked about. Instead, you would probably want to create a test that runs as a normal test and check the runtimes.

Regularly running performance tests that are set up with a helpful load level can provide a very useful feedback loop. However, there is a challenge with running performance tests regularly. How do you know if the slowdown you see is important enough to take action on or not? This is where benchmarking comes into play.

## Benchmarking

Knowing when to mark a performance test as failed is a bit of a black art. How slow is too slow? What if the slowness of the run was just a glitch? This gets even more difficult when you try to automatically decide if there is a failure during a test run.

One simple thing you can do is to set a high value for the failure time – two to three times what you would expect the maximum run time to be. This way, if there is a failure, you can be pretty sure that something quite serious has happened. However, this strategy isn't perfect. It will miss code slowdowns that are less dramatic but still user-impacting. It also runs the risk of allowing several small slowdowns to accumulate. In that case, when there finally is a slowdown that pushes you past the limit, it is hard to fix because it does not have just one root cause.

Another more complex strategy is to measure the runtimes for each run and compare the current runtime against those of previous runs. This requires a bit of tooling built around your run for storing runtimes. It also requires a strategy to compare the current run to previous runs. Do you want to see if it is slower than the average of the last three or five runs? Or do you want to look at a running average and make sure that its slope never gets above a given value? What levels do you set the increase at? To arrive at these answers, you will need to experiment to find a benchmark that will fail under adverse conditions but not constantly fail.

Benchmarking is the process of creating that line that defines how fast things should be. When you benchmark for a simple performance test with a single load level, you should keep your benchmark as simple as possible. However, when trying to create a benchmark for a test with a more complex load profile, you need to consider not just how slow things should be allowed to get but also how the time relates to the load.

In most applications, it is OK if things slow down a bit when it is under heavy load, but you will need to figure out what the maximum slowdown you can tolerate as the load increases is. You should also figure out what performance level you would want to tolerate at lower loads. Even if you are OK with your app getting a bit slower under times of extreme pressure, you still want it to be very snappy and responsive when it is under normal levels of load. When designing a performance benchmark, you should consider where it should be set at various load levels.

## Repeatability

Another important performance testing consideration is repeatability. With most regression tests, when we talk about repeatability, we mean that the test runs in the exact same way every time so that we can debug it if there is a failure. With performance testing, this idea of repeatability is relaxed a bit. When generating load with virtual users, Postman and other tools will not be able to run exactly the same requests at exactly the same times as they did in the previous run. Variations in the speed of the requests will cause variations in when the next request is run and, therefore, variations in the overall run. This means that if you run the same load profile twice against the same app with no code changes, you might see slightly different average response times and even a different number of errors.

However, despite these challenges, we still want a performance testing run to be repeatable. How do we get that? There are certain factors that we can't control, but we should make sure that we take care of all those that we can. One important thing is to standardize the test environment. Especially if your test environment is ephemeral (something like a Docker image that gets built on demand for the test environment), it is important to ensure that each time you spin up a test environment, you do so with the same settings. It should have the same underlying hardware, with the same number of cores and memory, etc. It should also have the same operating system with the same tools installed each time. There are infrastructure as code tools like Terraform that can help with this, but whether you are using something like that or just a simple script, make sure that you can create consistent testing environments. Don't set up or use a test environment that varies every time you run tests. This will just add noise to performance testing and make the already difficult task of benchmarking them even harder.

Test repeatability is also affected by the initial data setup in the test environment. If your database starts with nothing in it, you might get very different results than if it has thousands or millions of entries when you start your test. It is important that you start from a consistent point. This means that, most of the time, you don't want to do performance testing on a shared testing site that other users might also work on. You also don't want to run multiple performance tests against the same site at the same time.

You should also think about your strategy for performance tests to clean up after themselves. If you want to be able to run the same test again on the same environment, you need to make sure the data from the previous run is cleaned up before starting a new run. However, I would caution against always automatically deleting the data. Tracking down the cause of performance slowdowns can be tricky, and it is often helpful to have the data that was created left there for debugging purposes. If you are working with ephemeral test environments that you can create on demand, it is probably best to just create a new environment and use it to rerun a test instead of trying to reuse the same environment. If you do need to reuse an environment, make sure you have a cleanup script available that can get the app to a consistent starting point.

Another thing that can affect the repeatability of tests is the cold start problem. Often, the first couple of requests to a server will take a lot longer, as the server starts up some processes that might not have run yet. This cold-start data can bias your performance testing results. If the environment you test on is newly created, it would be a good idea to run a few warmup requests before starting the main performance testing run.

In this regard, it is helpful to write down a clear set of steps that you will use when setting up and running your performance tests. This list should include a data management strategy with things like what settings to use when creating the test environment, what test data it should be pre-populated with, and what, if any, commands should be run before kicking off the performance testing run. Ideally, you would automate as many as possible of these steps, but any that can't be automated should be written down in a playbook or checklist that you can follow to keep things consistent across runs.

## **Collaboration and communication**

Every aspect of software development requires this, but I still want to point out the importance of collaboration and communication in performance testing. A good performance test needs to know what kind of loads to use. You can make a guess at this on your own, but you should also solicit the opinions of those who sell or market the app to get an idea of how many users to expect.

You should also work closely with developers to figure out problems when they arise, as well as areas of the code that they are interested in getting feedback on.

Another important stakeholder in any performance testing endeavor is the system administrator or the operations engineer. These are the people who are responsible for operating the application in production, and they will have a lot of valuable information about what kind of hardware the system has and how to access metrics, like the CPU and memory usage.

As with any testing, if we find problems that stakeholders don't care about, we are not finding valuable problems. In performance testing, it is difficult to know where to set the benchmarks. Reach out to others in your company, particularly those with customer-facing roles, and try to agree on an understanding of what the needs are for your application. Working together with others is essential to creating good tests.

## Summary

Performance testing is a complicated sub-specialty in testing. When you have access to performance testing experts, engaging them to help you with your testing is a great idea. However, the reality is many companies do not have the resources to keep performance testing experts on staff. That is where you can step in and use some of the things you've learned in the chapter.

You've learned about some of the different kinds of performance testing. You learned how to think about processing, memory, and connection loads. You have learned about how to generate different kinds of loads and about the different load profiles that you can use. We covered spike, ramp, and endurance load profiles. We looked at what each of those loads are and how to generate them in Postman.

We then dug into performance testing loads in more detail as we covered how to run some simple performance tests in Postman. You learned how to create and use a load profile for a single request and how to do some more complicated testing with multiple requests. I also showed you how to understand some of the reports that Postman gives and how to debug failing requests.

There are many considerations to keep in mind when performance testing, some of which I discussed in this chapter. I talked about when you should do performance testing, how to benchmark your tests, and the importance of creating repeatable tests. I also shared some ideas with you that you can use in each of those areas. We also looked at the importance of working together in collaboration with others as you seek to create useful performance tests.

Performance testing has a lot of nuances to it, and although I certainly could not cover all the details of it in one small chapter, you should have a basic grasp of it after working through this material. You should be able to do some simple performance testing and hopefully provide your team with some useful insights.

As we come to the end of our API testing journey together, I hope that you have learned a lot along the way. We have come a long way in this book. We started with some of the basic theory and terminology of API testing, including the principle of API design. We also learned how to use OpenAPI specifications to guide the design and testing of APIs. We covered how to think about API test automation and how to design APIs.

We, of course, learned how to do this all in Postman and how to use many powerful Postman features, such as data-driven testing and workflow testing. We covered API authorization and how to create good test validation scripts. We looked at how to run tests in CI build environments and dug into how to monitor APIs in production. Testing existing APIs has its own set of considerations compared to testing a greenfield API, so we looked at things to keep in mind for that kind of testing.

We also discussed some more advanced API testing topics – things like creating and using mock servers to make your testing more efficient and using contract testing to keep consumers aligned with the API server development. We covered some of the basics of security and performance testing as well.

Along the way, I have taken you through challenges and examples that have helped you learn and apply these ideas. You really have learned a lot! As you finish this book, I want you to stop and take a moment to reflect on how far you have come. It is a big accomplishment, and you should feel confident in your ability to tackle any API testing challenges that come your way.

A learning journey is just that – a journey. Don't let your learning journey end at the end of this book. I have given you a good grounding, and if you have worked through the examples, you have been able to see a lot of the details of how to use Postman, but the more you use it in real-life scenarios, the more you are going to learn.

Keep using this tool. Keep reading about API testing. Keep testing and exploring, and never stop learning.

Happy testing!

## Leave a review!

Enjoyed this book? Help readers like you by leaving an Amazon review. Scan the QR code below to get a free eBook of your choice.







packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## **Why subscribe?**

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

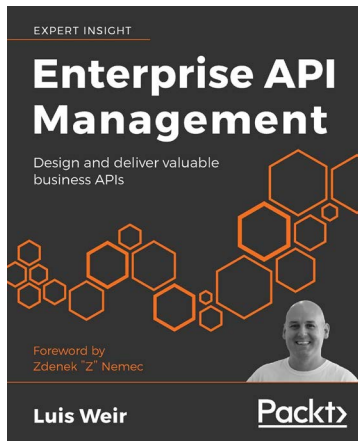
At [www.packt.com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.





# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



## **Enterprise API Management**

Philip Wilkins

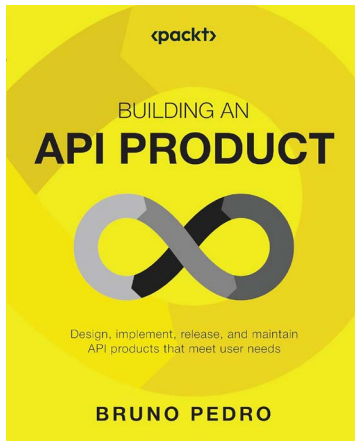
Luis Weir

Rolando Carrasco

ISBN: 978-1-78728-443-2

- Create API strategies to deliver business value
- Monetize APIs, promoting them through public marketplaces and directories
- Develop API-led architectures, applying best practice architecture patterns
- Choose between REST, GraphQL, and gRPC-style API architectures

- Manage APIs and microservices through the complete life cycle
- Deploy APIs and business products, as well as Target Operating Models
- Lead product-based organizations to embrace DevOps and focus on delivering business capabilities



## **Building an API Product**

Bruno Pedro

ISBN: 978-1-83763-044-8

- Master each stage of the API lifecycle
- Discover technologies and protocols employed in building an API product
- Understand the different API design definition and validation techniques
- Generate an API server from a machine-readable definition
- Understand how to set up and analyze API monitors
- Familiarize yourself with the different gateways for releasing an API
- Find out how to create an API portal that attracts users
- Gain insights into planning and communicating API retirement to users

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Share your thoughts

Now you've finished *API Testing and Development with Postman, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

# Index

## A

### Akamai EdgeGrid

- reference link 113
- using 113

### Amazon Web Services (AWS) 102

### API actions

- DELETE 9
- GET 9
- HEAD 9
- OPTIONS 9
- PATCH 9
- POST 9
- PUT 9

### API Blueprint 52

- reference link 52

### API bugs

- example 219
- finding 213-217
- service, resetting 218, 219
- testing 215-217
- tests, setting up 214

### API calls

- challenge 17
- idempotent call 3-5
- making 14
- making, to test application 16

- safe request 3-5
- test application, setting up 15
- types 3

### API consumer 257

### API description documents 51

### API description format 51

### API design example 42

- e-commerce API, designing 42, 43

### API documentation 36

- best practices 40, 41
- RESTful API Modeling Language (RAML) 42
- with Postman 37-40

### API keys

- using 101

### API request structure 7

- actions 9
- body 13
- endpoints 8, 9
- headers 11, 12
- parameters 10
- query parameters 10, 11
- request parameters 10
- response 13

### API responses

- after tests, cleaning up 131
- body response, checking 122
- checking 118, 119

- custom assertion objects, in Postman 127
- folder and collection tests, creating 130, 131
- headers, checking 127
- status code, checking 119, 120
- tests, creating 130

**API responses, body response**

- JSON properties, checking 123-126
- string usage, checking 122

**API responses, status code**

- Chai assertions, using in Postman 121
- pm.test method, using 120

**API schema 61-63**

- defining 51

**API security 94**

- authentication 95, 96
- authorization 95
- in Postman 96

**API specification 50, 51**

- languages 49
- terminology 51
- types 52
- using, in Postman 66

**API specification, types**

- API Blueprint 52
- OpenAPI 53
- RESTful API Modeling Language (RAML) 52
- Swagger 53

**API test automation 219**

- collection, setting up in Postman 220-226
- collection, sharing in Postman 238
- considerations 74
- environment, updating 226, 227
- example 222
- reviewing 220
- tests, adding to DELETE request 236, 237
- tests, adding to first request 228, 229

- tests, adding to POST request 232, 233
- tests, adding to PUT request 235, 236
- tests, adding to second request 229-231
- tests, cleaning up 233, 234
- tests, creating 221, 226
- working 237

**API testing considerations 17, 21**

- business problems 20
- exploration 17, 18
- exploratory testing case study 18-20

**API tests**

- types 75, 76

**API types 21**

- GraphQL APIs 26
- REST APIs 21
- SOAP APIs 22

**Application Programming Interfaces (APIs) 2**

- building 32, 33
- example 34, 35
- need for 33, 34
- persona 33

**assertion 117****authentication 276, 277****authorization 275, 277****automated testing 72-74**

- writing 74, 75

**AWS Region 102****AWS Signature**

- using 102, 103

**B****Basic Auth**

- using 98, 99

**bearer tokens**

- using 100

- Behavior-Driven Development (BDD)** 121
- benchmarking** 310, 311
- broken object-level authorization** 277, 278
- broken property-level authorization** 279
- built-in reporters**
  - Newman, using 186, 187
- business workflow** 164

## C

- callback function** 121
- Chai Assertion Library** 121
- Chai assertions**
  - using, in Postman 121
- CI/CD builds**
  - general principles, for using Newman 192
  - Newman, integrating into 192
- collection scope** 85
- connection load** 296, 297
- consumer** 258
- consumer-driven contracts** 258, 259
  - advantages 259
- continuous integration (CI)** 310
- contracts**
  - consumer-driven contracts 258
  - creators 258
  - provider-driven contracts 259
- contract tests** 255, 256
  - adding, to collection 264-267
  - collection, creating 261, 263
  - failures, fixing 271
  - Postman Interceptor, using 269, 270
  - running 267-271
  - setting up, in Postman 260
  - sharing 272, 273
  - using 257

- credentials**
  - handling, in Postman 114
- custom assertion objects**
  - .body 128
  - .header 128
  - in Postman 127
  - .json 128
  - .jsonBody 129
  - .responseSchema 129
  - .responseSize 129
  - .responseTime 129
  - .status 128
  - .statusCode 127
  - .statusCodeClass 127
  - .statusReason 128
  - .withBody 128
- custom reporter**
  - creating 188-192

## D

- data-driven inputs**
  - setting up 148
- data-driven testing** 145
  - creating, in Postman 150
  - defining 146, 147
  - outputs 148, 149
- data-driven testing, with multiple APIs**
  - challenge 157
  - challenge hints 157, 158
  - challenge, setting up 157
- data-driven tests**
  - running, in Newman 183, 184
- data input**
  - creating 150, 151
  - responses, comparing from file 154-156
  - test, adding 152, 153



**data scope** 86  
**denial-of-service attacks** 280  
**design-first methodology** 59  
**digest authentication** 109-111  
**draft state** 170

**Dynamic Data**  
    mocking 248-251

**dynamic variables**  
    reference link 253

## E

**e-commerce API**  
    actions, defining 44, 45  
    API design, modeling 47  
    designing 42, 43  
    endpoints, defining 43, 44  
    query parameters, adding 45, 46  
    RAML specification, using in Postman 46, 47  
**endpoint tests** 76  
**endurance load profile** 303  
**Entity Relationship Diagram (ERD)** 223  
**Envelope node** 23  
**environments**  
    using, in Newman 181-183  
**environment scope** 86  
**Equivalence Class Partitioning (ECP)** 145, 148  
**exploratory testing** 72-74  
**external reporters**  
    Newman, using 187

## F

**fixed load profile** 297-299  
**Flows feature**, in Postman

    building 167-171  
    used, for workflow testing 165, 166  
**fuzzing (fuzz testing)** 282, 283  
    tests, cleaning up 287-289  
    with built-in methods in Postman 290, 291  
    with Postman 283-287

## G

**Get Homeworld request** 134  
**GetTasks request** 228  
**GitHub Actions**  
    example 193-197  
**GitPod** 15  
**global scope** 84, 85  
**GraphQL** 3, 26  
**GraphQL APIs** 26  
    example 26, 27

## H

**Hawk authentication** 111, 112  
**htmlextra**  
    used, for generating reports 188  
**Hypermedia API** 19  
**Hyrum's Law** 259

## I

**idempotent call** 4  
**Imgur**  
    reference link 104  
**integration tests** 79

## J

**JavaScript Object Notation (JSON)** 123

**JSON Formatter**

reference link 285

**JSON Placeholder**

reference link 150

**L****linear workflows 162-164****load profiles**

endurance load profile 303  
fixed load profile 297-299  
ramp load profile 301, 302  
spike load profile 299, 301  
using, in Postman 297

**local scope 86, 87****M****maintainable tests**

creating 87  
logging, using 87  
test reports 88

**Markdown 52****memory load 295****mock servers 241, 242**

considerations 243  
private servers, using 251  
setting up, in Postman 244, 245  
third-party APIs, mocking 252, 253  
usage, scenarios 242, 243  
using 251  
working with 241

**Mock Server values**

creating 246, 247  
modifying 245, 246

**monitor results**

cleaning up 211

viewing 208, 210

**Mutation-based Fuzzing 283****N****Newman 88, 176**

data-driven tests, running 183, 184  
environments, using 181-183  
general principles, using in CI/CD builds 192  
installing 176  
installing, with npm 178  
integrating, into CI/CD builds 192  
running 178-181  
running, options 181-185  
setting up 176  
tests, reporting on 185  
using, built-in reporters 186, 187  
using, external reporters 187

**Newman HTML Reporter 187****New Technology LAN Manager (NTLM) 112****Node.js package manager (npm) 176**

installing 177  
reference link 177  
used, for installing Newman 178

**NTLM authentication**

using 112

**O****OAuth**

using 103, 104

**OAuth 1.0 108****OAuth 2.0**

access token 105-108  
application, registering 105  
setting up, in Postman 104, 105

**OpenAPI 53**

reference link 52

**OpenAPI Specification (OAS) 50**

- budgeting.yaml file 59, 60
- creating 53-59
- examples, using 64, 65
- info section 54
- parameters, defining 63
- paths section 55, 57
- Pet schema 58
- request bodies, describing 64
- response section 56

**OWASP API Security list 275**

- authentication 275, 277
- authorization 275, 277
- broken object-level authorization 277, 278
- broken property-level authorization 279
- unrestricted access to business workflows 281
- unrestricted resource consumption 280, 281
- unsafe consumption of APIs 281, 282

**OWASP API Security Project**

- reference link 275

**OWASP API Security Top 10**

- reference link 282

**P****parameter 150****parameterized testing 146****performance load types 294**

- connection load 296, 297
- memory load 295
- processing load 294, 295

**performance testing 76, 297**

- benchmarking 310, 311
- collaboration 312, 313
- communication 312, 313
- considerations 309

- need for 309, 310
- repeatability 311, 312
- running, in Postman 304, 305

**pm.test method**

- using 120

**Postman**

- additional monitor settings, using 203
- Akamai EdgeGrid, using 113
- API keys, using 101
- API security 96
- API specifications, using 65, 66
- authorization 97, 98
- AWS Signature, using 102, 103
- Basic Auth, using 98, 99
- bearer tokens, using 100
- Chai assertions, using 121
- contract tests, setting up 260
- credentials, handling 114
- data-driven testing, creating 150
- digest authentication 109-111
- fuzzing, with built-in methods 290, 291
- Hawk authentication 111, 112
- installing 5, 6
- load profiles, using 297
- mock server, creating 66, 67
- mock servers, setting up 244, 245
- monitor, creating 200-202
- monitor, setting up 200
- multiple requests, running 306-308
- NTLM authentication, using 112
- OAuth 2.0, setting up 104, 105
- OAuth, using 103, 104
- performance testing, running 304, 305
- reference link 97
- request, saving 7
- request, setting up 6, 7
- request, validating 68

- starting 6
- tests, adding to monitor 206-208
- used, for fuzz testing 283-287

**Postman, additional monitor settings**

- email notification for run
  - errors, receiving 203
- email notification for run failures, receiving 203
- enable SSL validation 206
- follow redirects 205, 206
- retry if run fails 204
- set delay between requests 205
- set request timeout 204, 205

**Postman API key 266****Postman environments**

- using 140
- variables, managing 140, 141

**Postman Interceptor**

- using, in contract tests 269, 270

**Post-response section 131****pre-request scripts**

- data, passing between tests 133-135
- request workflows, building 135, 136
- setting up 132
- variables, using 133

**private servers**

- using 251

**processing load 294, 295****provider 258****provider-driven contracts 259****Q**

- query parameters 10, 11, 36

**R**

- ramp load profile 301, 302

- random input 283

- repeatability 311, 312

**repeatable tests**

- creating 89

**reports**

- generating, with htmlextra 188

**Representational State Transfer (REST) 3****request parameter 10****request workflows**

- building 135, 136
- looping 136-138
- running, in collection runner 138, 139

**RESTful API 21**

- Wikipedia link 21

**RESTful API Modeling Language (RAML) 42, 52**

- reference link 42

**Route Parameters**

- mocking 247, 248

**S**

- schema 51

- security testing 76

**Send Request block**

- configuring 166, 167

**Simple Object Access Protocol (SOAP) 2, 22****smoke tests 224****SOAP APIs 22, 23**

- example 23, 24

- spike load profile 299, 300

- Swagger 52, 53

**Swagger Editor**

- reference link 53

## T

**table-driven testing** 146

**Test-Driven Development (TDD)** 121

**test method** 120

**test organization** 79, 80

- collection variable 81

- environment variable 80, 81

- variable scope, selecting 82

- variables, using 87

**tests**

- cleaning up 287-289

- reporting, on Newman 185

**test structure** 77

- creating 77, 78

**third-party APIs**

- using 252, 253

**Time Addition**

- reference link 157

**Time Subtraction**

- reference link 157

## U

**Uniform Resource Locator (URL)** 8

**usable API**

- creating 35

- error messages 36

- structure 35, 36

## V

**variables**

- using 87

**variable scope**

- collection scope 85

- data scope 86

- environment scope 86

- global scope 84, 85

- local scope 86, 87

- selecting 82

- working 82-84

## W

**w3 primer**

- reference link 22

**Web Services Description Language (WSDL) file** 23

**workflow tests** 161

- API calls, checking 172, 173

- complex things, checking 171, 172

- creating, guidelines 171

- with Flows feature in Postman 165, 166

**workflow tests, types** 161

- business workflow 164

- linear workflow 162-164

## Y

**YAML specification**

- reference link 52

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781804617908>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

